DA 9 A 99 ANNI

Andy Hunt

Imparare a programmare in Java con Minecraft

"I ragazzi oggi crescono con l'idea della programmazione come arte (al pari della musica), non come un passatempo da nerd."
– Nick Grantham

Imparare a programmare in Java con Minecraft

Andy Hunt

APOGEO

Un grazie davvero speciale a mio figlio Stuart, che mi ha suggerito di scrivere questo libro e ha risposto a un sacco di mie domande stupide su Minecraft. Grazie anche al resto della mia famiglia per avermi sopportato quando sparivo sotto le cuffie e digitavo come un pazzo in un mondo immaginario.

Grazie al mio editor Brian Hogan, al managing editor Susannah Pfalzer, al production manager Janet Furlow e a tutti quelli di Pragmatic Bookshelf per avermi aiutato a terminare questo lavoro in tempo.

Grazie ai revisori tecnici, compresi Said Eloudrhiri, Ingo Haumann, Jack H., Dan Kacenjar, Andrés N. Robalino e Zachary Thomas. Un grazie in più a Joshua McKinnon per i suoi controlli dettagliati e attenti.

Un ringraziamento particolare allo staff di CanaryMod, e soprattutto a Jason Jones, per il loro sostegno.

Iniziamo da qui

Benvenuto!

Grazie per aver scelto questo libro. Spero che non lo troverai noioso e che ti divertirai a leggerlo. Se non ti è mai capitato di scrivere un programma prima d'ora, non preoccuparti. Partiremo dall'inizio e faremo un passo alla volta. Non serve che tu abbia già esperienza.

Minecraft piace a tutti. Uno dei motivi del suo successo è che permette a chiunque di partecipare alla realizzazione del gioco. Ti dà infatti la possibilità di costruire e fabbricare. Che sia un rifugio d'emergenza in modalità Sopravvivenza, una simulazione di pietrarossa o il tuo castello, potrai creare qualcosa da solo.

A volte, però, le funzioni che il gioco ti mette a disposizione non sono sufficienti. Potresti volere delle mucche che volano mentre vanno a fuoco, per esempio, o avere la possibilità di rinchiudere un nemico in una gabbia di roccia. Per conquistare queste e molte altre capacità, dovrai aggiungerle tu al gioco.

Le applicazioni sul tuo computer o sul tuo smartphone sono scritte in un testo speciale chiamato *linguaggio di programmazione*. Un linguaggio di programmazione non è complesso da imparare come le lingue che si parlano nel mondo (come il cinese o il tedesco) ma è diverso da quello che usi tutti i giorni per scrivere e comunicare.

I linguaggi di programmazione sono tantissimi. Alcuni sono famosi ma poco potenti. Altri vengono usati solo da poche persone molto preparate, ma sono incredibilmente solidi e molto più difficili da capire.

Minecraft è scritto nel linguaggio di programmazione chiamato Java, che è piuttosto potente ma che in alcune sue parti è abbastanza complicato e può creare confusione. Qui ti spiegheremo i suoi elementi di base, i più facili, e tralasceremo quelli più difficili.

In questo libro ti insegneremo in poco tempo a programmare in Java. Alla fine, ne saprai abbastanza da poter svolgere le operazioni più comuni di questo linguaggio e creare da solo i tuoi plug-in di Minecraft. (Se non sai cos'è un plug-

in, te lo spieghiamo tra poco.)

Vedremo anche come impostare il tuo server di Minecraft, creando sul cloud delle copie di riserva del codice che scrivi, mentre verso la fine daremo un’occhiata ad alcune tecniche di programmazione avanzate.

Questo libro fa per te?

Questo libro è fatto per i lettori che non hanno alcuna esperienza di programmazione ma che vogliono intervenire in Minecraft da veri esperti. Se non hai familiarità con il gioco, puoi trovare un aiuto guardando i numerosi video in Internet o leggendo alcuni libri che ne parlano.

NOTA

Se non sei ancora molto esperto di Minecraft, se hai dei dubbi o se vuoi avere una panoramica completa sul gioco e sui suoi elementi, puoi visitare il wiki in lingua italiana all'indirizzo http://minecraft-it.gamepedia.com/Minecraft_Wiki.

Se però sei qui, suppongo che tu sia già un fan entusiasta di Minecraft, e che tu voglia imparare qualcosa sulla sua programmazione.

Se hai meno di 8-9 anni o se hai qualche difficoltà a capire come si programma in Java, puoi fare una piccola "deviazione" e provare prima a lavorare con un linguaggio più semplice.

Per esempio, Scratch e la sua versione più recente, Snap!, sono due piccoli linguaggi che mostrano attraverso le immagini come si combinano i vari elementi della programmazione, e che possono aiutarti a imparare i concetti di base (vai agli indirizzi http://scratch.mit.edu e http://snap.berkeley.edu).

NOTA

In questo libro troverai dei riferimenti a siti o libri che purtroppo si trovano solo in inglese. Se non conosci questa lingua, dovrai farti aiutare a interpretarli.

Una volta che avrai acquisito un po' di familiarità con l'argomento, un linguaggio basato su testo come Java ti sarà più comprensibile.

Se invece sai già come fare, per partire ti basterà avere un computer con installato Windows, OS X o Linux.

Cosa serve per iniziare

Minecraft è un'applicazione *client-server*, cioè divisa in due parti.

La prima parte è il *client*, cioè l'applicazione che gira sul tuo computer, portatile o desktop. Il client mostra le immagini del mondo di Minecraft e accetta i tuoi comandi quando ti sposti e agisci nel gioco.

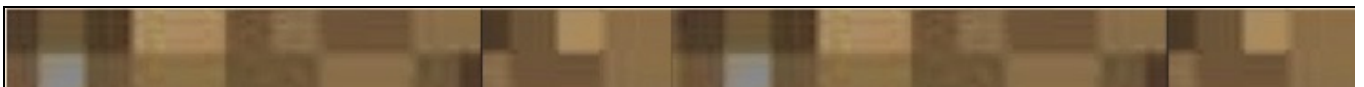
La seconda parte è il *server*, che tiene traccia di tutto quello che avviene nel gioco, compresi i giocatori che sono connessi, i loro inventari, quello che hanno costruito, dove si trovano e così via. La maggior parte del tempo il server lavora su una macchina dall'altra parte del paese, ma potrebbe girare anche sul tuo computer.

Il client e il server parlano tra loro attraverso Internet, esattamente come fai tu quando usi un browser web per collegarti e giocare online o vedere un po' di foto di gatti.

Per aggiungere o modificare una funzionalità in Minecraft, devi aggiungere o modificare il programma Java sul server. Ecco quello che imparerai in questo libro: programmare in Java scrivendo istruzioni (che chiameremo "codice sorgente", solo "codice" oppure "il programma") per creare dei plug-in per il server di Minecraft. Un *plug-in* non è altro che un pezzo di codice che aggiungi a un programma esistente. (Alcuni scrivono delle modifiche di Minecraft, chiamate *mod*, all'interno del client grafico, ma qui non ce ne occuperemo.)

Prima di iniziare a lavorare con i plug-in, devi impostare un server Minecraft locale, che ti servirà per fare i test, e installare il linguaggio di programmazione Java e un paio di altre applicazioni. Ci occuperemo di tutto questo nei primi capitoli. Installare tutti questi elementi non è un gran divertimento, anzi, è proprio noioso, ma cercheremo di sbrigarcela in fretta.

Per aiutarti a tenere una traccia dei tuoi progressi, alla fine di ogni capitolo trovi un paragrafo chiamato "La tua toolbox" che riporta una barra con la percentuale di nuove conoscenze che hai acquisito.

Partirai da zero…

… e finirai con un bel 100%:

In alcuni capitoli andremo più rapidamente che in altri, e in alcuni troverai più informazioni che in altri, ma farai sempre e comunque dei progressi.

Niente panico

Visto che avremo tanto di cui parlare ma poco spazio per farlo, prima ti mostrerò le cose e come usarle e poi te le illustrerò nel dettaglio. A volte ti sentirai come se ti avessero buttato in una piscina senza fondo: sarai disorientato, farai fatica a trovare la direzione, ma non preoccuparti. Da lì a poco ti spiegherò tutto. Potrà capitare che tu riesca a usare qualche caratteristica con successo anche se in realtà non hai capito come funziona.

Questo capita anche nella vita di tutti i giorni: puoi accendere una lampada e utilizzarla senza sapere come viene generata l'elettricità. Volendo, potresti anche costruirne una senza avere la minima idea di com'è fatto un generatore. Quello che ti serve sapere è solo come i vari pezzi stanno insieme.

Ed è proprio su come assemblare le varie parti che ci concentreremo.

Se ti serve aiuto

Su Internet trovi uno spazio dedicato a questo libro con domande, risposte, aggiornamenti e suggerimenti. Vai alla pagina https://pragprog.com/book/ahmine2 e fai clic sulla scheda *Discuss*.

Scarica i file degli esempi

Dalla stessa pagina, fai clic su *Source code* per scaricare tutto il codice sorgente del libro; il link esatto è https://media.pragprog.com/titles/ahmine2/code/ahmine2-code.zip. Scaricalo sul Desktop del tuo computer (utilizzerai il Desktop per la maggior parte del lavoro, ma ne parleremo meglio in seguito).

Quello che stai scaricando è un file archivio compresso, quindi dovrai decomprimerlo. Se lavori in OS X o Linux puoi farlo dalla riga di comando; se invece lavori con Windows puoi usare WinZip o 7-Zip (disponibili agli indirizzi http://www.winzip.com/it/downwz.html o http://www.7-zip.org).

Fatto? Bene!

Man mano che andremo avanti, conoscerai nuovi strumenti o imparerai nuovi modi per impiegarli. Terremo traccia di tutto questo nella “toolbox” – la tua cassetta degli attrezzi – alla fine di ogni capitolo. Arrivati in fondo al libro, la tua toolbox sarà abbastanza ricca da permetterti di progettare i tuoi plug-in e il relativo codice da zero!

Alcune regole

Il codice o i comandi che ti mostro come esempio appaiono così:

```
$ Ho digitato questa riga come esempio per te.
```

Il codice o i comandi che invece dovrai digitare ti appaiono in grassetto:

`$ questa parte devi digitarla tu (ma senza il segno del dollaro)`

Le parole in *corsivo* sono dei segnaposto, quindi non devi digitarle direttamente. Per esempio, in una riga come:

```
me.chat(string msg);
```

devi sostituire solo la parte in corsivo, così:

```
me.chat("Stanno arrivando dei Creeper");
```

Pronto? E allora iniziamo!

Comanda il tuo computer

Uno dei primi e più bei giochi per computer nel quale veniva costruito un mondo da esplorare è stato Colossal Cave Adventure (http://it.wikipedia.org/wiki/Colossal_Cave_Adventure), creato nel 1976.

Era un gioco d'avventura basato solo sul testo, senza immagini. Per dare le istruzioni bastava digitare delle frasi semplici, per esempio "vai a nord", "prendi l'ascia" o "uccidi il troll", e il gioco faceva quello che gli si chiedeva, compreso strangolare il troll a mani nude.

Anche oggi i giochi utilizzano dei comandi di testo, Minecraft compreso. Ti sarà senz'altro capitato di digitare dei comandi nella finestra della chat usando un carattere `/`.

In questo capitolo imparerai a impartire dei comandi al tuo computer in un modo molto simile, usando la *riga di comando* per costruire dei plug-in e lavorare con i file.

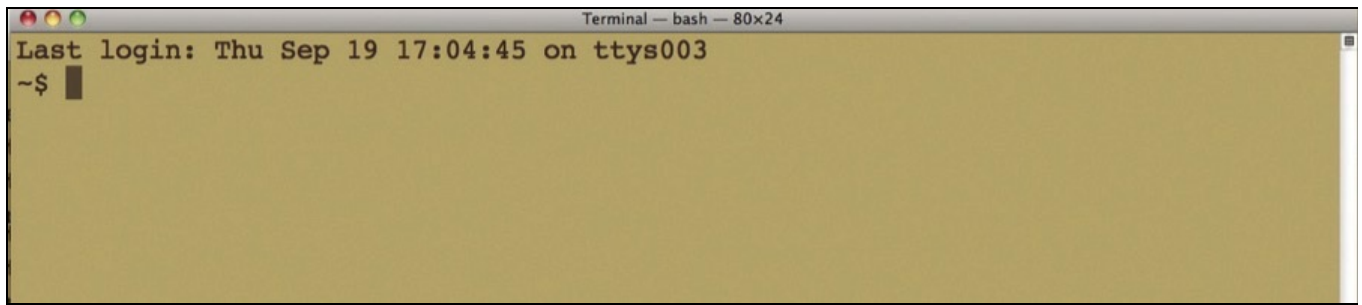
La riga di comando è uno strumento potente che ti permette di lavorare sul tuo computer così come su computer lontani, per esempio sul cloud. Parleremo meglio di come farlo nell'Appendice D, "Come installare un server sul cloud".

Puoi anche usare un elaboratore a riga di comando per scrivere i programmi; l'elaboratore contiene un proprio linguaggio di programmazione, separato da Java. Per darti una mano, ho preparato uno *script* (cioè un elenco di comandi eseguibili) che ti aiuterà a costruire e installare i plug-in. Lo vedremo tra poco.

Se sai già usare la riga di comando, puoi passare direttamente alla fine del capitolo.

Usare la riga di comando

Sul mio computer, la riga di comando ha l'aspetto che vedi in questa figura. Sul tuo computer potrebbe avere dei colori e dei caratteri diversi, e comunque puoi sempre impostarla secondo i tuoi gusti. Io preferisco le lettere nere su uno sfondo marroncino.

```
Last login: Thu Sep 19 17:04:45 on ttys003
~$
```

Ci sono vari modi per accedere alla riga di comando, che sono leggermente diversi a seconda del sistema operativo con cui lavori.

Windows ha una riga di comando molto essenziale, che si apre con il comando `cmd.exe`. Se la tua versione di Windows ha un comando *Start*, puoi selezionare *Start* > *Esegui* e poi digitare `cmd.exe`, oppure puoi usare la casella di ricerca che si apre per trovare ed eseguire `cmd.exe`. Il mio consiglio è però quello di non usare `cmd.exe` da solo; a questo proposito, leggi le istruzioni nel riquadro "Installare BusyBox in Windows" più avanti nel capitolo.

Su OS X, puoi aprire la riga di comando selezionando *Applicazioni* > *Utility* > *Terminale*.

Se usi Linux, è molto probabile che tu sappia già come accedere alla riga di comando. Per completezza, comunque, hai tre possibilità: aprire una shell, avviare la console o fare clic con il tasto destro mouse sul Desktop e aprire il terminale.

(Fortunatamente, tutte queste fastidiose differenze tra Windows, OS X e Linux spariscono una volta che inizi a scrivere codice Java; questo linguaggio di programmazione, infatti, si comporta nello stesso modo su tutte le piattaforme.)

Una volta aperta l'applicazione della riga di comando, puoi iniziare a digitare i comandi nell'elaboratore, chiamato *shell*. I comandi che utilizzerai sono pochi e semplici; te li spiegherò man mano.

Ogni shell visualizza un breve messaggio, chiamato *prompt*, che indica che è

pronta per accettare i comandi che immetti. Tuttavia, invece di un messaggio chiaro come "Digita i tuoi comandi, maestro. Sono pronto", i prompt che vedrai sono un po' più misteriosi.

Windows mostra in genere qualcosa come `C:\>`. I sistemi Linux e OS X potrebbero mostrare `$` o `%`. Inoltre i prompt potrebbero includere altre informazioni, come il tuo nome, quello del computer o quello di una cartella (*directory*). Poiché ogni sistema è diverso, negli esempi del libro ho scelto il più semplice e ti mostrerò il prompt dei comandi come `$`. Indipendentemente dall'aspetto del prompt sul tuo computer, quando vedi `$`, è lì che devi immettere i comandi.

NON DIGITARE IL SEGNO DEL DOLLARO (`$`). È solo un simbolo che uso per indicarti dove devi digitare, ma non devi mai includerlo in quello che scrivi:

```
$ digita qui, ma non inserire il segno del dollaro
```

Quando hai finito di scrivere un comando e vuoi che il computer lo esegua, premi il tasto Invio. Questa era la parte facile. Adesso devi imparare qualche comando di base!

Per ottenere un elenco di file e cartelle, si usa il comando `ls`:

```
$ ls
```

Per passare a una cartella diversa da quella corrente, si usa invece il comando `cd`. Per esempio, per passare alla cartella *Desktop*, digita

```
$ cd Desktop
```

e verrai trasportato a *Desktop* (o a qualsiasi altra cartella che hai indicato). Quando la shell si apre per la prima volta, ti trovi in una specie di cartella predefinita, di partenza, chiamata *home* ("casa"). Per vedere quali file contiene, digita il comando `ls` (che sta per *list files*, ovvero "elenca file") e premi Invio.

Nei sistemi Windows che usano `cmd.exe`, devi invece digitare `dir`. Mentre il resto del mondo usa il carattere `/` nei nomi delle cartelle, Windows usa una barra che va nella direzione opposta (`\`), chiamata *backslash*.

Queste differenze spiegano perché in Windows è consigliabile usare il bash della shell standard POSIX, come ti spiego tra poco nel riquadro "Installare BusyBox in Windows".

Vedrai apparire un elenco di file. Digita `cd Desktop`, e ti troverai nella cartella *Desktop*. Digita il comando `ls`, e vedrai un elenco dei file di *Desktop*. Visto? Stai

lavorando con la riga di comando! Andiamo avanti.

Spostarsi tra le cartelle

Quando vuoi vedere i file sul disco fisso del tuo computer, in genere usi dei programmi “grafici”, come Esplora risorse in Windows o il Finder su OS X. L’idea è quella di mostrarti i file, le cartelle e le applicazioni/programmi sul computer in modo che tu possa spostarti tra di essi e lavorare.

Qui faremo la stessa cosa, ma in modo più potente e senza immagini. Se sai già di cosa parlo, puoi saltare direttamente alla fine del capitolo. Altrimenti continua a leggere.

INSTALLARE BUSYBOX IN WINDOWS

Gli ambienti OS X e Linux sono basati su Unix, che ha un ambiente a riga di comando e una shell molto ricchi. Per Unix esiste uno standard, chiamato POSIX, che include i comandi e le caratteristiche del linguaggio. La shell dello standard POSIX è in realtà un linguaggio di programmazione vero e proprio, che permette di scrivere degli script per compiere delle operazioni di base sul computer.

Windows non è però così sofisticato. L’elaboratore dei comandi predefinito non fa granché, i comandi hanno nomi diversi e i nomi di cartelle e file sono specificati in modo differente. Per ottenere un po’ di omogeneità, consiglio sempre agli utenti Windows di scaricare BusyBox. Questo software installa una shell più professionale e alcuni comandi comuni compatibili con POSIX, gli stessi usati da OS X, Linux e dal resto mondo. Per installare la shell e i comandi di base per Windows, scarica sul Desktop il file `ftp://ftp.tigress.co.uk/public/gpl/6.0.0/busybox/busybox.exe`.

Una volta scaricato il file, rinominalo da `busybox.exe` a `sh.exe`. Poi apri una finestra `cmd.exe` e digita quanto segue:

```
C:\> cd Desktop
C:\> C:\Windows\system32\cmd.exe /c sh.exe -l
```

Vedrai apparire una nuova shell con un prompt con il segno di dollaro. È da qui che lanceremo i comandi e compiremo le nostre azioni. Nota un dettaglio importante: `sh.exe` ha un flag `-l` che dice alla shell di agire come una “shell di login” nella quale puoi digitare i comandi.

Per comodità puoi tenere un file batch sul Desktop da dove lanciare la shell. Per farlo, crea un file di testo e salvalo sul Desktop. Chiama il file `shell.bat` e digita questa riga di testo al suo interno:

```
C:\Windows\system32\cmd.exe /c sh.exe -l
```

Salva il file. Ora puoi fare doppio clic su `shell.bat` e ottenere una shell compatibile con POSIX. Fatto questo, puoi accedere a tutti i comandi che utilizzeremo (come `ls`, `mv`, `cp` e `pwd`) e specificare i nomi delle cartelle usando il carattere / invece del \ di Windows. In questo modo, tutto funzionerà nella stessa maniera in Windows, OS X e Linux, e potrai anche usare gli script di comando che troverai nel libro per creare e installare i plug-in.

L’home page di BusyBox all’indirizzo http://intgat.tigress.co.uk/rmy/busybox/index.html contiene ulteriori informazioni su BusyBox per Windows.

L'insieme dei file e delle cartelle sul tuo computer è chiamato *file system*. In qualsiasi momento, la finestra del Finder o di Esplora risorse sul Desktop o della shell dei comandi mostra la cartella corrente.

Sul Desktop puoi avere aperte anche più finestre, una per ciascuna cartella sul disco.

Tutte le finestre di shell aperte funzionano nello stesso modo: a ogni finestra corrisponde una *cartella corrente*. Alcuni sistemi sono impostati in modo da mostrarti la cartella corrente al prompt.

Tuttavia, puoi sempre capire in quale cartella ti trovi digitando il comando `pwd` (che sta per *print working directory*, cioè "mostra cartella di lavoro"):

```
$ pwd
/Users/andy/Desktop
```

Prova così: apri una nuova shell e, prima di fare qualsiasi altra cosa, digita `pwd` al prompt (qui è mostrato come `$`; sul tuo computer potrebbe apparire diversamente):

```
$ pwd
```

Questo comando ti indica la *cartella home*, che è il punto da cui partirà ciascuna delle tue shell.

In ogni shell, tutti i comandi che lanci vengono eseguiti secondo questa idea di cartella corrente: molti dei programmi che usi partono da qui per eseguire, aprire e salvare i file.

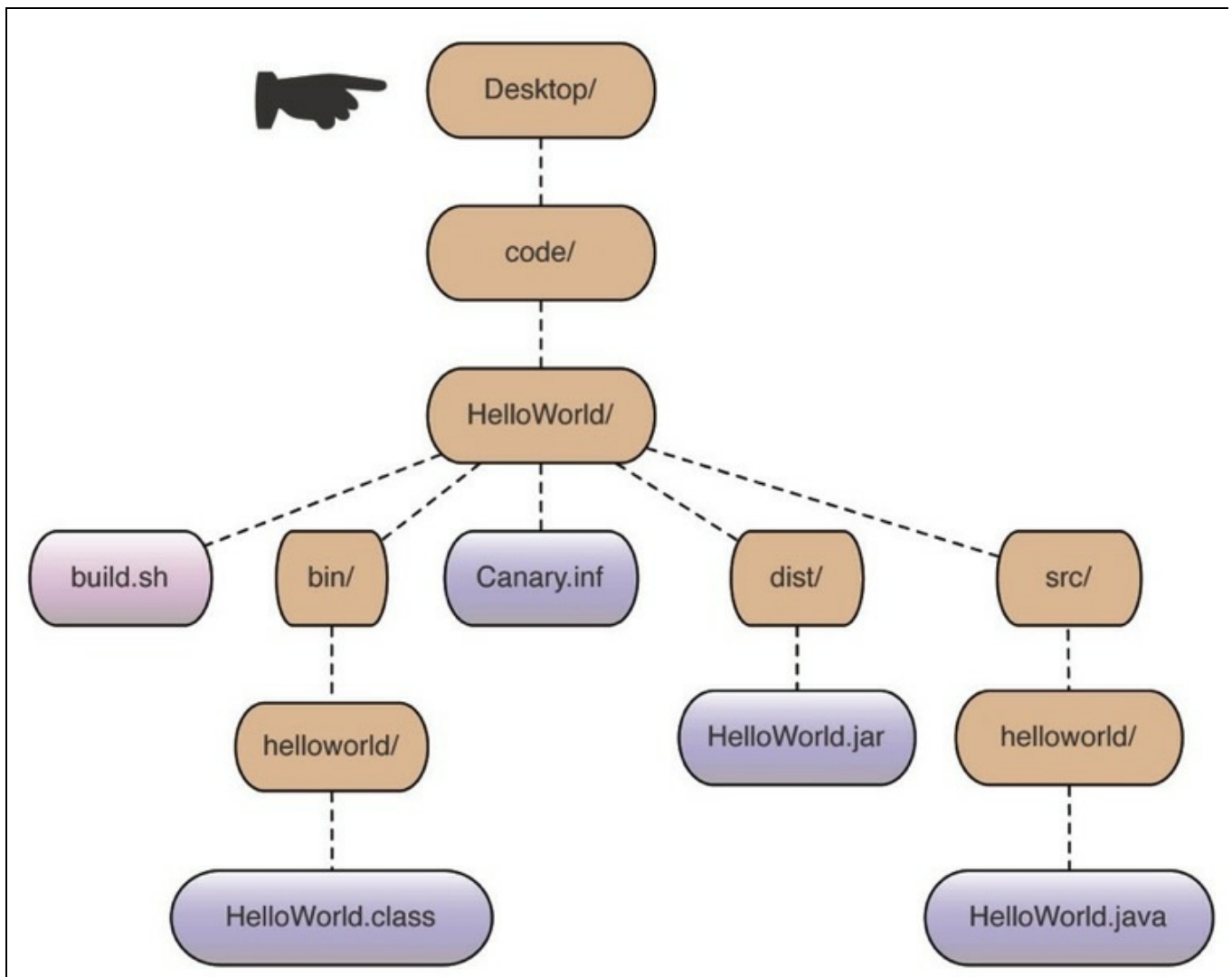
Se non l'hai ancora fatto, scarica sul Desktop i file del libro (http://media.pragprog.com/titles/ahmine2/code/ahmine2-code.zip) e decomprimi l'archivio (usa `unzip` dalla riga di comando oppure, in Windows, usa WinZip o 7-Zip). Il file verrà decompresso in una cartella chiamata *code*, che contiene tutti gli esempi del libro.

Sotto a *code* si trova un gruppo di cartelle di plug-in, una per ciascun plug-in del libro. Inizieremo dando un'occhiata ai file di *HelloWorld*. Questa cartella contiene altri file e sottocartelle.

Le figure che seguono ti mostrano come spostarti tra le cartelle.

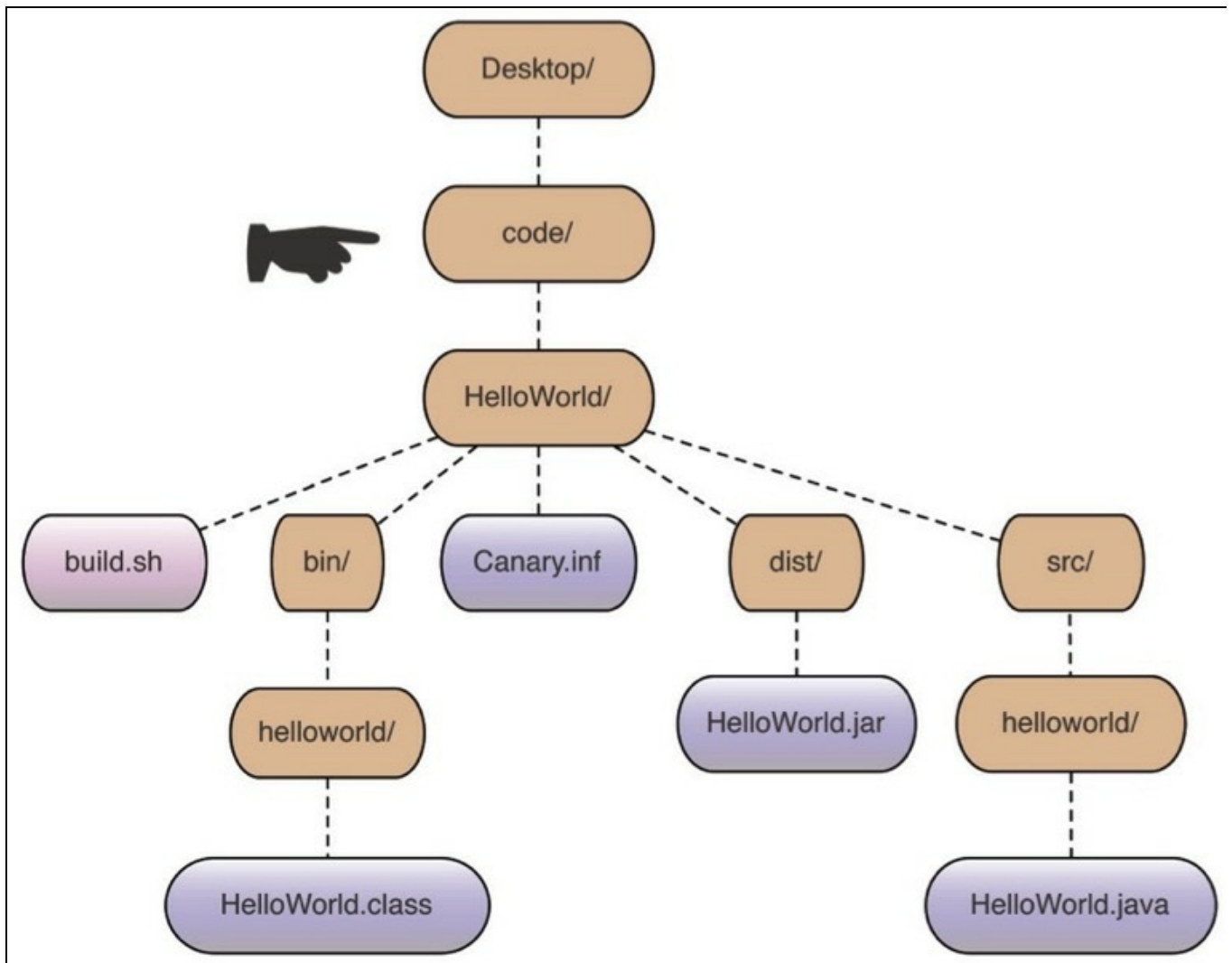
Partendo da qui:

```
$ cd Desktop
```

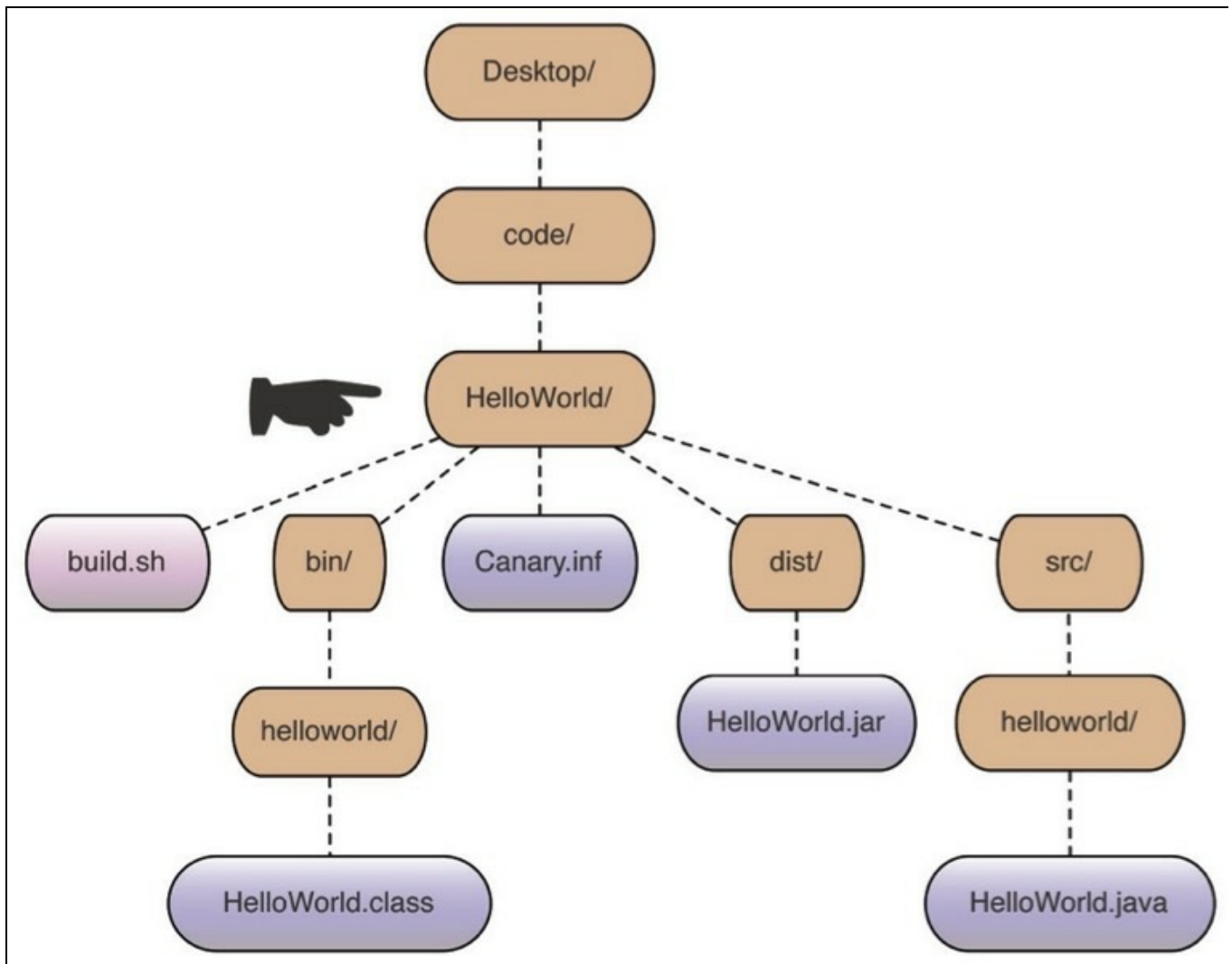
ti trovi qui:

(Se non funziona, prova con `cd ~/Desktop`, oppure vai alla fine del capitolo per qualche suggerimento.)

Ora scendi nella cartella *code* digitando

```
$ cd code
```

Ti troverai qui:

Entra in *HelloWorld*:

```
$ cd HelloWorld
```

… e ti troverai qui:

Ora elenca i file che si trovano in questa cartella. Vedrai questo:

```
$ ls
Canary.inf bin/ build.sh dist/ src/
```

Il mio sistema (OS X) è configurato in modo da mostrare le cartelle con uno slash (una barra, `/`) alla fine. (Se per te non è così, prova a digitare `ls -F`.) In questo esempio la mia cartella corrente contiene due file e altre tre cartelle. I tipi di file sono spesso identificabili dall'ultima parte del nome, il *suffisso*. Qui c'è uno script di shell con un suffisso `.sh` e un file di configurazione con un suffisso `.inf`. La cartella *src/* contiene una sottocartella *helloworld*; qui si trova un file sorgente Java con un suffisso `.java` (imparerai di più sui tipi di file man mano che proseguiremo).

Per entrare nella cartella *src*, digita `cd src`:

```
$ cd src
$ ls
```

```
helloworld
```

Scendi ancora più in profondità, in *helloworld*:

```
$ cd helloworld
$ ls
HelloWorld.java
```

Nella cartella *src/helloworld* si trova il file `HelloWorld.java`, il cuore del nostro primo plug-in.

Sei arrivato fino a *HelloWorld/src/helloworld*. Ma come fare per tornare a *HelloWorld*? Per retrocedere di un livello digita questo:

```
$ cd ..
```

Per tornare indietro di due livelli digita questo:

```
$ cd ../..
```

Hai visto i due punti? Un punto (.) indica la cartella corrente; con `cd`, non serve a molto, ma con altri comandi sì, soprattutto quando si devono copiare dei file.

Ora immagina di dover accedere a una cartella che non si trova in quella corrente o in quella immediatamente sopra. Come fare? Supponi di trovarti in un punto completamente diverso, per esempio in */home/minecraft*, e di dover arrivare a */Users/andy/Desktop/code/HelloWorld*.

Dovrai scrivere qualcosa del genere:

```
$ cd /Users/andy/Desktop/code/HelloWorld
```

Lo slash è quello che fa la differenza. In precedenza, quando hai digitato `cd src`, il comando `cd` ha cercato *src* sotto la cartella corrente. Ma se avessi digitato `cd /src`, avresti cercato una cartella chiamata *src* sotto una cartella chiamata /, la cosiddetta *root*.

La cartella root è quella che si trova in cima a tutto il file system. È a capo del codice, del Desktop, di tutto. Sotto alla root si trovano la cartella *home* e il Desktop. Nel mio caso il percorso è */Users/andy/Desktop*. Volendo, potrei arrivare in questo punto digitando questa sequenza di comandi:

```
$ cd /
$ cd Users
$ cd andy
$ cd Desktop
```

Ma come vedremo tra poco esiste un metodo molto più rapido. Parlando di scorciatoie, infatti, non serve neanche specificare il nome per intero.

Sulla maggior parte dei sistemi, esiste una scorciatoia da tastiera che permette di abbreviare la digitazione dei nomi più lunghi: è il tasto Tab, ⇥. Se digiti le

prime lettere di un nome lungo e poi premi Tab, il nome si completerà automaticamente. Supponi di essere nella tua cartella *code*:

```
$ ls
Adventure             CreeperCow       LocationSnapshot    SquidBomb
ArrayAddMoreBlocks    EZPlugin         MySimple            SquidBombConfig
ArrayOfBlocks         FireBow          NameCow             Stuck
BackCmd               FlyingCreeper    NamedSigns          install
BackCmdSave           HashPlay         PlayerStuff         mkplugin.sh
BuildAHouse           HashPlayClamp    PortingGuide.txt    numbers
CakeTower             HelloWorld       Simple              runtime
CanaryLinks.txt       LavaVision       Simple2
CowShooter            ListPlay         SkyCmd
```

Se digiti `cd Ad` e poi premi Tab (o abbastanza lettere perché il nome sia distinguibile), il nome si completerà automaticamente

```
$ cd Adventure/
```

e ti basterà premere Invio. Nel caso di nomi brevi questo trucco non serve granché, ma se hai una cartella dal nome lunghissimo, come *RumpelstiltskinReincarnationSpellPlugin*, digitare `Ru` e poi premere Tab diventerà molto utile!

COPIA E INCOLLA

Può accadere che tu debba copiare del testo e incollarlo nella riga di comando. Per esempio, potresti voler copiare una riga da questo libro e poi incollarla.

Copiare e incollare dalla riga di comando può essere leggermente diverso che farlo in un'applicazione come Word o in un browser web. Per iniziare, devi comunque fare clic sul testo con il mouse e trascinare per selezionarlo.

In Linux, nella maggior parte delle applicazioni, puoi premere i tasti Ctrl+C per copiare e Ctrl+V per incollare. Se si trovi nella riga di comando, potresti dover aggiungere anche il tasto Maiusc; quindi, per copiare, premi Ctrl+Maiusc+C.

Su OS X, usa CMD+C per copiare e CMD+V per incollare.

La finestra della riga di comando di Windows è leggermente diversa. Per prima cosa devi abilitare la *Modalità modifica rapida*. Fai clic con il tasto destro del mouse sulla barra in alto nella finestra della shell e seleziona *Proprietà*. Nella scheda *Opzioni*, nella sezione *Opzioni di modifica*, spunta la casella *Modalità modifica rapida* per attivarla.

Per utilizzare questa modalità, dopo aver selezionato il testo, premi Invio per copiare e poi fai clic con il tasto destro del mouse o premi Ctrl+V per incollare.

Questo espediente vale solo per la riga di comando. Altrove nel sistema (il tuo editor di testo e così via), usa sempre Ctrl+C e Ctrl+V, come al solito.

Prova da solo

Creiamo ora qualche cartella e file attraverso la riga di comando. Imparerai a

creare la tua copia di un plug-in, con le cartelle e tutto il resto.

Parti dalla cartella *Desktop* (per essere certo di essere lì, digita `pwd`) e crea una nuova cartella chiamata *myplugins* usando il comando `mkdir`:

```
$ cd Desktop
$ pwd
/Users/andy/Desktop
$ mkdir myplugins
```

Elenca i file contenuti in *Desktop* (usando `ls`), e vedrai tutti i file racchiusi in questa cartella, oltre che una nuova cartella, *myplugins*. Entra in *myplugins*:

```
$ cd myplugins
```

Usa `pwd` per avere una conferma che ti trovi in *myplugins*.

Se esegui qui il comando `ls` ma non ti appare niente, è solo perché non abbiamo ancora creato dei file. Rimediamo realizzando la struttura della cartella, che sarà la stessa di quella del plug-in *HelloWorld*. Inizia creando una cartella che si chiamerà come il plug-in:

```
$ mkdir HelloWorld
```

Ora esegui `cd` per scendere in profondità in *HelloWorld*:

```
$ cd HelloWorld
```

Adesso puoi creare tutte le cartelle che ti servono: *src*, *src/helloworld*, *bin* e *dist*:

```
$ mkdir src
$ mkdir src/helloworld
$ mkdir bin
$ mkdir dist
```

Usa `ls` per assicurarti che ci siano:

```
$ ls
bin/ dist/ src/

$ ls src
helloworld/
```

Qui ti servono tre file, che puoi copiare dal codice di esempio del libro. Puoi trascinarli e rilasciarli utilizzando la consueta finestra grafica, oppure puoi usare il comando per copiare, `cp`:

```
$ cp ~/Desktop/code/HelloWorld/build.sh .
```

Il carattere della tilde (`~`) sta per "la mia cartella *home*." E ricorda: devi usare il punto! La riga di comando completa significa "copia questo file nella cartella corrente".

Ti serve anche il file che ti indico qui sotto, quindi copialo:

```
$ cp ~/Desktop/code/HelloWorld/Canary.inf .
```

Hai così creato le cartelle e i file di supporto necessari per un plug-in.

I FILE SUL DESKTOP

In questo libro faremo tutto il lavoro sul Desktop perché è il punto più facile dove trovare i file, e vale tanto in Windows quanto su OS X e Linux.

Nella vita quotidiana, è improbabile che tu voglia riempirlo con tutti i nuovi progetti su cui stai lavorando, ma finché non avrai acquisito dimestichezza su come muoverti e impostare gli ambienti, rimani sul Desktop.

Se non funziona

Una situazione che potrebbe crearti problemi è quella in cui il nome della tua cartella *home* contiene degli spazi. Per esempio, se lavori in Windows e ti chiami "Mario Rossi", digitando un comando usando la tilde

```
$ cp ~/Desktop/code/HelloWorld/build.sh .
```

otterrai qualcosa del genere:

```
$ cp C:/Users/Mario Rossi/Desktop/code/HelloWorld/build.sh .
```

Il computer interpreta la riga come se dicesse "copia `C:/Users/Mario` e `Rossi/Desktop/code/HelloWorld/build.sh` nella cartella corrente", e otterrai un errore del tipo "nessun file o cartella" ("no such file or directory").

Le soluzioni sono due. Puoi usare un percorso relativo digitando `..` per le cartelle genitore, così da salire di "due livelli" ed entrare in *code*:

```
$ cp ../../code/HelloWorld/build.sh .
```

Oppure puoi digitare il tutto direttamente racchiudendo il nome del file tra virgolette:

```
$ cp "C:/Users/Mario Rossi/Desktop/code/HelloWorld/build.sh" .
```

Un altro problema che potresti incontrare è quando non sei nella cartella in cui credevi di trovarti. Se hai un dubbio, puoi sempre ricorrere al comando `pwd` per visualizzare la cartella di lavoro corrente:

```
$ pwd
/Users/andy/Desktop
```

Qui sono nella cartella *Desktop*, il punto da cui inizierà la maggior parte del nostro lavoro.

Partiamo dalla cartella Desktop

Sui quasi tutti i sistemi è possibile digitare `cd Desktop` per accedere alla cartella *Desktop*. Se non funziona, potresti dover digitare `cd ~/Desktop` – con la tilde – oppure potresti dover scrivere tutto per esteso, come in `cd /Users/andy/Desktop`.

Qualunque sia il metodo utilizzato, quando ti dico di iniziare da *Desktop* o semplicemente di digitare `cd Desktop`, dovrai farlo sempre, a prescindere da dove ti trovi in quel momento, come:

```
$ cd Desktop
```

dalla tua cartella *home*, innanzitutto:

```
$ cd
$ cd Desktop
```

usando una tilde per la cartella *home*:

```
$ cd ~/Desktop
```

digitando il nome per esteso della tua cartella *home*:

```
$ cd /Users/mario/Desktop
```

o racchiudendo il tutto tra parentesi se il nome contiene degli spazi:

```
$ cd /Users/”Mario Rossi”/Desktop
```

Quello che vedrai sarà sempre

```
$ cd Desktop
```

E ora divertiamoci

Nel codice del libro che hai scaricato, nella cartella *Desktop/code*, trovi una sottocartella speciale chiamata *Adventure*. Usando la riga di comando, digita `cd` e dai un’occhiata ai file e alle cartelle che contiene.

Puoi utilizzare `ls` per elencare le cartelle come abbiamo fatto qui. Per vedere rapidamente i file di testo (quelli con il suffisso `.txt`), puoi usare il comando `cat`.

Parti dalla cartella *Desktop* (in uno dei modi sopra descritti):

```
$ cd Desktop
$ cd code
$ cd Adventure
$ cat README.txt
These are some files to make exploring the file system a little more fun.
```

Esegui il comando `ls`, guarda l’elenco che appare ed esplora un po’ le sottocartelle. Scopri i tesori – e i pericoli – che contengono.

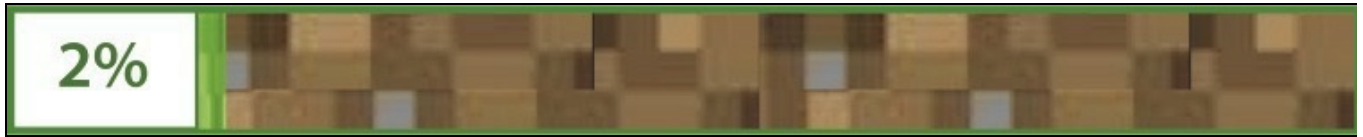
Comandi più comuni

Ecco alcuni dei comandi più comuni che utilizzerai nella shell.

`java`	Esegue le classi e gli archivi Java (JAR) come un programma.
`javac`	Compila il codice sorgente Java nei file `.class`.
`cd`	Cambia la cartella.
`pwd`	Mostra la cartella di lavoro corrente.
`ls`	Elenca i file nella cartella corrente.
`cat`	Visualizza il contenuto di un file.
`echo`	Visualizza un testo; con il prefisso `$`, mostra anche le variabili d’ambiente.
`mkdir`	Crea una nuova cartella.
`cp`	Copia un file.
`mv`	Sposta un file.
`rm`	Cancella definitivamente un file. Usalo con molta attenzione; non è uguale al Cestino e non hai la possibilità di fare Annulla.
`chmod`	Modifica i permessi dei file (compresi quelli di lettura, scrittura ed esecuzione).
`.`	(Un punto) Indica la cartella corrente.
`..`	(Due punti) Indica la cartella genitore.
`~`	(Tilde) Indica la tua cartella home.

Per continuare

Adesso devi imparare a digitare nel codice sorgente Java. Ti servirà Java e un'applicazione che metta insieme il tutto. Scaricheremo questi due elementi nel prossimo capitolo, e poi passeremo a creare dei plug-in.

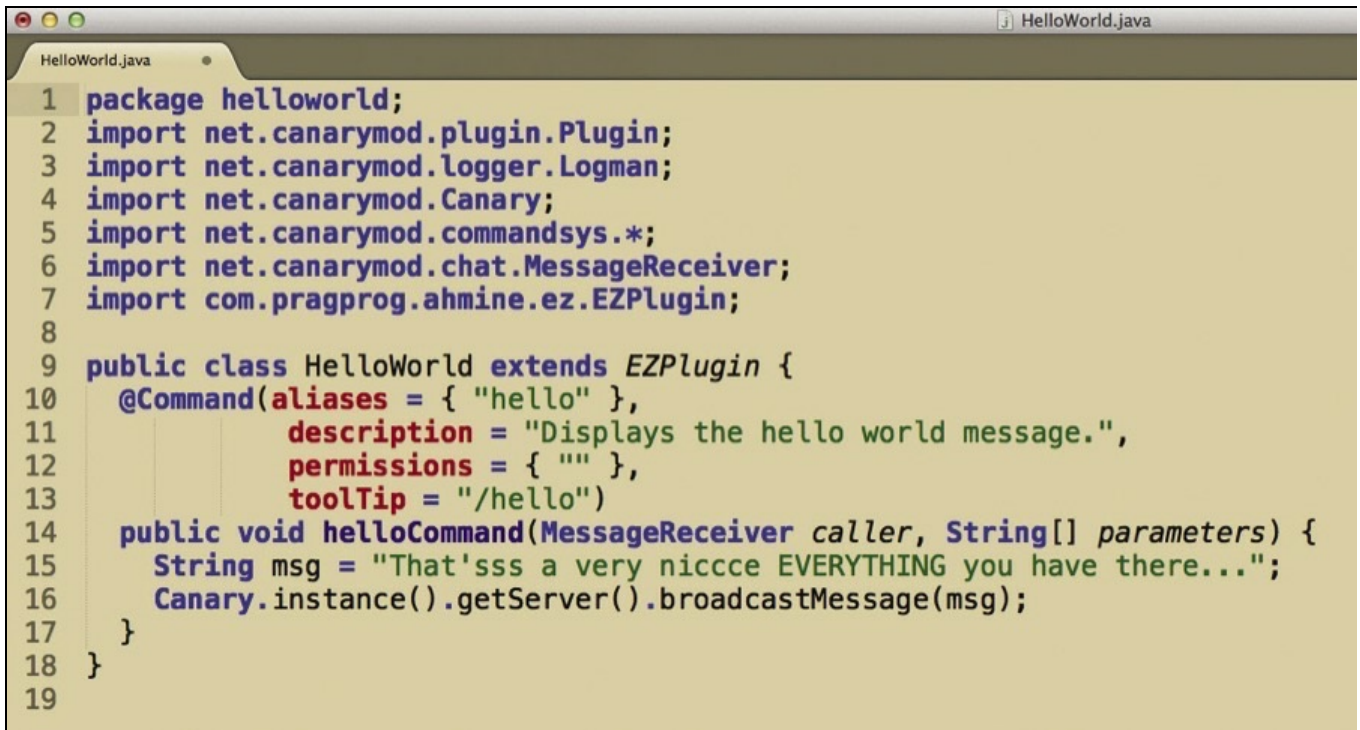
La tua toolbox

Adesso sai come:

- Usare la shell a riga di comando.

Capitolo 2

Aggiungere un editor e Java

Se vuoi scrivere programmi in Java, ti serve qualcosa per farlo, uno strumento per modificare i file di testo di questo linguaggio. Potresti utilizzare degli editor di base come Blocco note o TextEdit, ma sarebbe una sofferenza!

Non puoi neanche ricorrere a un software come Word o un altro elaboratore di testi simile. Queste applicazioni non sono nate per la programmazione, e non salvano i file in un formato utilizzabile da Java: i file prodotti da Word sono pieni di font, colori e di altre informazioni di formattazione.

Quello che ti serve è un editor di testo concepito per programmare. Ne ho in mente uno buono, te ne parlerò tra poco. Dovrai anche installare Java per costruire i plug-in e farli funzionare, e ovviamente ti servirà Minecraft. Una volta installato tutto questo, sarai pronto per il prossimo capitolo, quando realizzerai il primo plug-in.

Queste pagine potrebbero contenere alcuni comandi nuovi (e probabilmente misteriosi) e alcuni concetti che creano confusione. Non preoccuparti se non li conosci o non li capisci; non sono cose che si incontrano tutti i giorni quando si scrivono i plug-in; sono però un piccolo male necessario da affrontare se vuoi installare quello che ti serve.

Partiamo dall'editor.

Installare un editor per scrivere il codice

I plug-in e i programmi si scrivono in un miscuglio di inglese e di punteggiatura strana. Molte delle cose che digiterai saranno poco più lunghe di un messaggio, ma meno di un tema di quelli che fai a scuola. Per scrivere i programmi ti serve un editor fatto apposta. Il mio consiglio è quello di usare Sublime Text, che vedi in questa figura.

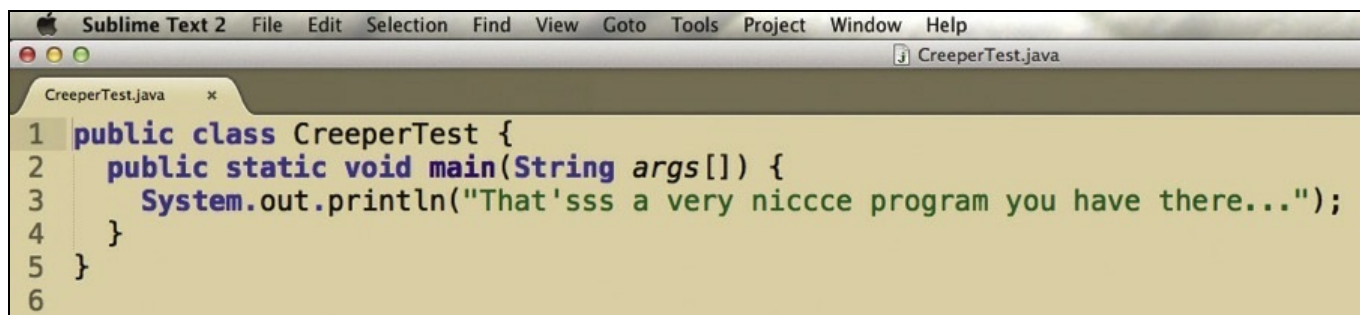
```
HelloWorld.java
1  package helloworld;
2  import net.canarymod.plugin.Plugin;
3  import net.canarymod.logger.Logman;
4  import net.canarymod.Canary;
5  import net.canarymod.commandsys.*;
6  import net.canarymod.chat.MessageReceiver;
7  import com.pragprog.ahmine.ez.EZPlugin;
8
9  public class HelloWorld extends EZPlugin {
10   @Command(aliases = { "hello" },
11             description = "Displays the hello world message.",
12             permissions = { "" },
13             toolTip = "/hello")
14   public void helloCommand(MessageReceiver caller, String[] parameters) {
15     String msg = "That'sss a very niccce EVERYTHING you have there...";
16     Canary.instance().getServer().broadcastMessage(msg);
17   }
18 }
19
```

Sublime Text può essere scaricato da http://www.sublimetext.com e lavora in Windows, OS X e Linux. Puoi provare la versione gratuita (che non ha limiti di tempo, ma che ogni tanto potrebbe darti qualche seccatura) o mettere da parte un po' di soldi e comprarlo. Se preferisci utilizzare un editor diverso di cui magari hai sentito parlare, fallo pure. In Windows, per esempio, potresti procurarti l'editor gratuito Notepad++(http://download.html.it/software/notepad/). La scelta è molto personale; non esiste una risposta, dimensione o forma "giusta". L'importante è lavorare con un editor che abbia le funzioni necessarie.

Uno dei vantaggi principali di un editor costruito per i programmatori è l'*evidenziazione della sintassi*. L'editor riconosce le diverse parti del linguaggio Java, e mostra funzioni, variabili, stringhe e parole chiave in colori e caratteri diversi, come nella figura di prima.

Questo tipo di aiuto visivo può esserti molto utile quando stai imparando cosa vogliono dire i vari elementi. Nei prossimi capitoli te li spiegherò uno per uno; intanto però scegli il tuo editor e installalo. (Volendo potresti utilizzare anche IntelliJ IDEA o Eclipse; sono ambienti di sviluppo completi che includono un editor e il supporto di costruzione, ma per i principianti possono essere difficili da padroneggiare.)

Avvia Sublime Text (o l'editor che hai scelto). Se ti è capitato di usare un programma come Word, noterai delle somiglianze: c'è una barra dei menu in alto, e nel menu *File* trovi alcune voci utili come *New*, *Open* e *Save*.

Digita del testo, magari un paio di righe di un breve storia dell'orrore o qualcosa del genere ("Il riflesso nello specchio gli fece l'occhiolino"). Fai clic e trascina per selezionare il testo e premi il tasto Canc per eliminarlo. Fino a qui niente di diverso da un normale elaboratore di testi.

Prova da solo

Adesso devi assicurarti di poter creare un file, modificarlo e salvarlo. Non sei ancora pronto per un plug-in completo, quindi partiamo da un semplice file di prova.

Dalla barra dei menu dell'editor seleziona *File* > *New File* per aprire una finestra di testo vuota in cui digitare.

Prima di iniziare a scrivere, salva il file dandogli un nome. Seleziona *File* > *Save As* e salva il file come `CreeperTest.java` sul Desktop. Devi salvarlo proprio scritto così, con la C e la T in maiuscolo e il resto delle lettere in minuscolo. In questo modo il tuo editor saprà che stai per digitare del codice Java.

Scrivi il codice che vedi nella figura:

Sublime Text 2 File Edit Selection Find View Goto Tools Project Window Help

CreeperTest.java

```
1 public class CreeperTest {
2   public static void main(String args[]) {
3     System.out.println("That'sss a very niccce program you have there...");
4   }
5 }
6
```

Copialo esattamente così, iniziando da `public class` fino all'ultima `}`, compresa tutta la punteggiatura, le maiuscole, le minuscole e gli spazi. Non scrivere invece i numeri di riga, perché quelli li genera l'editor. Usa il tasto Canc (o il tasto Backspace, ←) per cancellare gli eventuali errori che compi mentre digiti.

Ora seleziona *File* > *Save* per salvare il tuo file di testo. Urrà! Hai un file con del codice sorgente Java!

Il tuo computer, però, non ha la minima idea di cosa farsene. È solo un file di testo, in fondo. Per quanto ne sa lui, potrebbe essere la ricetta di un dolce, un tema o un elenco di trucchi di gioco. Più avanti risolveremo questo problema.

Adesso ti serve Java.

I PERCORSI IN WINDOWS

Ricorda: lavorando dalla shell di BusyBox, le cartelle usano uno slash (`/`), mentre nel resto di Windows si utilizzano i backslash (`\`). Quindi, quello che in Windows e nei programmi di installazione è

```
C:\Utenti\il tuo nome\Desktop\server
```

per noi è

```
C:/Utenti/il tuo nome/Desktop/server
```

Installare il linguaggio di programmazione Java

Java non è solo un'applicazione: è più di una. C'è il compilatore, javac, l'esecutore dei programma, java, e l'utility di archiviazione, jar. Javac prende il tuo file di testo e crea un insieme chiamato *file class*. Puoi eseguire il comando `java`, dirgli di usare quel file, e il tuo programma prenderà vita. Vedremo il processo nel dettaglio più avanti, ma il succo è questo.

Java potrebbe già essere installato sul tuo computer. Prova a eseguire il compilatore, javac, e controlla:

```
$ javac -version
javac 1.7.0
```

Nel mio caso era già installato nella versione 1.7. Se così non fosse stato, avrei ottenuto questo messaggio:

```
$ javac -version
bash: javac: command not found
```

Volendo puoi anche controllare che sia nella stessa versione di javac (dovrebbe essere così, ma non si sa mai…):

```
$ java -version
java version "1.7.0_67"
Java(TM) SE Runtime Environment (build 1.7.0_67-b01)
Java HotSpot(TM) 64-Bit Server VM (build 24.65-b04, mixed mode)
```

Se devi installare Java, ti servirà la Java Development Kit Standard Edition (JDK SE) nella versione 7 o in una versione successiva; puoi scaricarla all'indirizzo http://www.oracle.com/technetwork/java/javase/downloads/index.html.

A seconda del tuo sistema operativo e del produttore, potresti trovare Java già installato, potresti avere java ma non javac o potresti avere una versione leggermente diversa. Per lavorare con gli esempi di questo libro ti serve Java JDK 1.7 o una versione successiva.

Indipendentemente dall'installer che utilizzerai, ti suggerisco vivamente di accettare tutte le risposte predefinite che ti vengono proposte, specialmente per quanto riguarda il punto dell'installazione. Java e i programmi a esso associati possono diventare un po' capricciosi e schizzinosi quando non trovano le cose dove se le aspettano.

Scarica l'installer del Java Development Kit (JDK) e segui le istruzioni.

In base alla tua piattaforma, potresti dover aggiungere la cartella *bin* di JDK al tuo “percorso”. Vedremo cosa vuol dire e come farlo nel prossimo paragrafo.

Intanto facciamo una prova e vediamo se funziona.

Prova da solo

Una volta installato JDK, puoi eseguire il programma di prova che hai scritto. Anche in questo caso si tratta di un programma semplice che può far contento Java. Non è ancora un plug-in di Minecraft, ma è un punto di partenza. Se funziona, sarai pronto per iniziare a creare i plug-in.

Esegui `cd` per il Desktop e assicurati che ci sia il file `CreeperTest.java`:

```
$ cd Desktop
$ ls
CreeperTest.java
```

Ora avvia il compilatore di Java su `CreeperTest.java` eseguendo il comando `javac` (che sta, appunto, per *Java compiler*):

```
$ javac CreeperTest.java
$ ls
CreeperTest.class CreeperTest.java
```

Il programma javac prima controlla per assicurarsi di aver capito tutto quello che hai digitato nel file `CreeperTest.java`. Se non trova errori, crea il file binario `.class`.

In realtà la maggior parte delle volte ci saranno degli sbagli. Non preoccuparti se vedrai apparire tonnellate di messaggi di errore: non vuol dire che hai distrutto il computer. I linguaggi di programmazione come Java sono noti per essere pignoli per tutto quanto riguarda maiuscole, minuscole, punteggiatura e tutto quello che si può inserire in un file.

Resta calmo (un consiglio che è sempre valido quando si lavora con i computer).

Cerca di interpretare il messaggio e ricontrolla il codice confrontandolo con il nostro esempio di poco fa. Nelle prossime pagine affronteremo altri problemi simili, e nell’Appendice A troverai alcuni consigli.

NOTA

Se niente di tutto questo ti aiuta, prova a visitare lo spazio di discussione sul Web dedicato al libro, dove altre persone come te hanno incontrato queste difficoltà. Il forum di discussione si trova all’indirizzo https://forums.pragprog.com/forums/382.

Una volta che javac termina il suo lavoro con successo e senza errori, puoi passare all'esecuzione.

Puoi chiedere a Java di eseguire il tuo programma passandogli il nome del file senza il suffisso `.class`:

```
$ java CreeperTest
That'sss a very niccce program you have there...
```

Ehi, funziona! Congratulazioni. Hai compilato ed eseguito il tuo primo segmento di codice. Puoi passare al paragrafo sull'installazione di Minecraft più avanti nel capitolo.

Se invece non dovesse funzionare, leggi il prossimo paragrafo; cercheremo di capire cos'è andato storto.

Se non trovi un comando Java

L’errore più probabile che potresti incontrare è “javac not found” (cioè “non ho trovato javac”) oppure “java not found” (“non ho trovato java”), che significa che anche sei hai installato Java, la shell non riesce a trovare le applicazioni `java.exe` o `javac.exe`.

I comandi che abbiamo usato finora erano già presenti o sono stati installati dall’installer di BusyBox. Ma quando installi Java, il computer potrebbe non sapere in che punto l’hai messo.

Quando digiti il comando `javac`, il computer deve trovare un file eseguibile chiamato `javac` (in Windows è `javac.exe`; su OS X è solo `javac`). Per cercarlo, va a guardare in un paio di posti standard. In Windows, potrebbe essere una cartella come *C:\Windows\System32*, mentre su OS X/Linux possono essere più cartelle, come
/usr/bin, */usr/local/bin* e così via.

Poiché il sistema ha i propri comandi con i quali è bene non fare confusione, e visto che dovrai aggiungere i tuoi per l’esecuzione, finiscono per esserci troppi posti in cui il computer dovrà cercare. Puoi dirgli esattamente dove guardare procurandogli un elenco di nomi di cartelle. Si parla di *percorso di ricerca*, o semplicemente *percorso* (*path*).

Per vedere cosa contiene il percorso della finestra della tua shell digita questa riga:

```
$ echo $PATH
```

Perché l’elaboratore della shell/riga di comandi trovi un comando da eseguire, deve trovarsi in una cartella inclusa nel percorso di ricerca. Dovrai quindi aggiungere la cartella *bin* di Java al tuo percorso.

La posizione della cartella dipende dal tuo sistema operativo e dall’installer che hai utilizzato. In Windows, una posizione tipica potrebbe essere *C:\Programmi\Java\jdk1.7.0_67\bin* (i tuoi numeri di versione potrebbero essere leggermente diversi).

Su OS X, viene quasi sempre installata in */usr/bin* o */usr/local/bin*, che sono entrambe già nel tuo percorso, ma la cartella potrebbe trovarsi in un punto del tutto diverso, per esempio in*/opt/local/java*.

Una volta trovata la cartella di installazione per Java, vedrai che conterrà una cartella *bin*. In *bin* troverai `java`, `javac`, `jar` e un po' di altre cose. Ora devi aggiungere il percorso completo per la cartella *bin* al percorso (`PATH`) della tua shell.

Il `PATH` è un elenco di cartelle separate da due punti (un punto e virgola in Windows). Per modificare il `PATH` per la shell bash che stiamo usando, segui i prossimi passi. (Se questo metodo ti provoca dei problemi, in particolare in Windows, dai un'occhiata alla pagina http://www.java.com/it/download/help/path.xml per avere un aiuto.)

1. Nella shell, spostati nella tua cartella *home* digitando `cd`. Conferma il percorso completo della tua cartella *home* digitando `pwd`.
2. Usando il tuo editor di testi, crea o modifica il file di avvio in questa cartella. Solitamente si tratta di un file chiamato `.profile` o `.bash_profile` (nota il punto che precede il nome). In Windows con BusyBox, dovrai utilizzare `.profile`. Negli altri casi, usa `.bash_profile`. In genere `ls` non mostra i file con il punto iniziale, ma questo accade con `ls -a` se il file esiste già. Potresti doverlo creare da zero; nessun problema.
3. Aggiungi una riga al file per modificare l'impostazione di `PATH`, aggiungendo la cartella *bin* di Java separata da due punti (`:`), o da un punto e virgola (`;`) in Windows.

Per esempio, su Linux/OS X, se il mio JDK fosse stato installato in */opt/local/java*, avrei dovuto aggiungere a `.bash_profile` la riga

```
export PATH="$PATH:/opt/local/java/bin:"
```

In Windows, devi modificare i backslash in slash e utilizzare il punto e virgola invece dei due punti; quindi, se Java è installato in *C:\Programmi\Java\jdk1.7.0_67\bin* (secondo lo stile di Windows), aggiungi a `.profile` una riga che dice (secondo lo stile di POSIX):

```
export PATH="$PATH;C:/Program Files/Java/jdk1.7.0_67/bin;"
```

Salva il file e chiudilo, quindi chiudi e riapri le finestre delle righe di comando per selezionare le nuove impostazioni. Devi chiuderle una a una e poi riaprirle se vuoi che tutto funzioni.

Per controllare il tuo percorso e vedere se la nuova impostazione è stata applicata, digita questa riga:

```
$ echo $PATH
```

Dovresti vedere una nuova voce che include la cartella *bin* di Java.

Altre ragioni per le quali non funziona

Vediamo ora qualche altra cosa che può andare storta anche se `PATH` è impostato correttamente. Assicurati di essere nella cartella giusta, digita `ls` e controlla che il file `CreeperTest.java` sia lì.

Fai attenzione a scrivere `javac CreeperTest.java` (con la parte `.java`), altrimenti potresti veder apparire un messaggio misterioso come questo:

```
error: Class names, 'CreeperTest', are only accepted if
annotation processing is explicitly requested
```

Se il comando `javac` riporta un errore sulla "syntax" (la sintassi) o un errore del tipo "not found" (non trovato) o "not defined" (non definito), significa che non ha capito il testo contenuto nel file `CreeperTest.java`, quindi è probabile che tu abbia sbagliato a scrivere qualcosa. Questo tipo di errori potrebbe apparire così:

```
CreeperTest.java:1: class, interface, or enum expected
```

Oppure potresti incontrare altri messaggi. Il numero racchiuso tra i due punti (`:1:`) è il numero di riga dove si trova l'errore.

Se non riesci a trovare l'errore, prendi una copia nuova del file dal codice sorgente del libro che hai scaricato (`code/install/CreeperTest.java`) e provalo.

Se il comando `java` non trova `CreeperTest.class`, accertati che il comando `javac` sia stato eseguito correttamente e che abbia prodotto con successo un file `.class`. Dovresti trovarti nella stessa cartella di quando `java` è in esecuzione.

Se vedi questo errore

```
Exception in thread "main" java.lang.NoClassDefFoundError: CreeperTest/class
```

è probabile che tu abbia digitato accidentalmente `java CreeperTest.class` (con la parte `.class` alla fine) invece di `java CreeperTest` (senza suffisso). Per ricapitolare, questi sono i comandi da compilare e poi eseguire:

```
$ javac CreeperTest.java
$ java CreeperTest
```

Ossia, devi specificare il suffisso `.java` durante la compilazione ma non devi digitare la parte `.class` durante l'esecuzione di `java`. Verifica anche se c'è un'impostazione per `CLASSPATH` (che è come `PATH` ma per le classi Java):

```
$ echo $CLASSPATH

$
```

Deve essere vuota. Se non lo è, accertati che contenga almeno un punto (.) per includere la cartella corrente.

Se neanche questo funziona, non spaventarti. Chiedi aiuto in giro: la programmazione spesso è il risultato di un lavoro di squadra.

Fatto! Questa era la parte più difficile.

Ora che Java è operativo, devi installare le parti di Minecraft.

Installare il client e il server di Minecraft

Minecraft è un sistema client-server, quindi ti serviranno tutti e due gli elementi: il client grafico che userai per giocare e il server a cui connetterti, dove aggiungeremo i plug-in.

Installare il client grafico di Minecraft

Probabilmente hai già questo elemento sul tuo computer, ma se così non fosse, scarica l'installer di Minecraft da `http://minecraft.net`. Segui le istruzioni di installazione, ma per adesso non avviarlo.

Quando giochi a Minecraft, il "client" è l'applicazione in esecuzione, che si collega al tuo account all'indirizzo `http://minecraft.net`.

Esistono anche client per i dispositivi iOS di Apple e per Android, compresi gli smartphone, ma sono un vicolo cieco; al momento in cui scrivo, non sono in grado di connettersi ai normali server di Minecraft e quindi di utilizzare i plug-in che creeremo.

Il client gestisce le immagini e i suoni, e ti permette di digitare le istruzioni di gioco. Per adesso non c'è ancora un server locale a cui connettersi, però, quindi è meglio se scarichiamo il codice del server.

Installare il server di CanaryMod

Adesso divertiamoci. Aggiungeremo dei plug-in al nostro server, quindi ti serve CanaryMod, uno speciale server di Minecraft creato dai tipi in gamba di CanaryMod.net apposta per utilizzare i plug-in.

Crea sul Desktop una cartella chiamata *server* (che si troverà in una posizione tipo */Users/iltuonome/Desktop/server*):

```
$ cd Desktop
$ mkdir server
```

Questa sarà la tua cartella del server, ed è qui che installerai le parti che lo compongono.

Il progetto CanaryMod ci sta tutto in unico file JAR, `CanaryMod.jar`, che viene usato per eseguire il server di gioco e anche per sviluppare nuovi plug-in.

Vai sul Web e visita la pagina del progetto CanaryMod all'indirizzo `http://canarymod.net/releases`. Vedrai un elenco di altre release raggruppate in base al numero corrispondente della versione di Minecraft. Io utilizzo l'ultima release, attualmente Minecraft 1.7.10. Tu dovrai scaricare il file JAR più recente. Per tua comodità, ne ho inclusa una copia nei file del libro.

Ogni file JAR può essere denominato con dei numeri in più relativi alla versione (qualcosa come `CanaryMod-1.7.10-1.1.0.jar`; i tuoi numeri potrebbero essere anche più lunghi). Scarica il file nella cartella *Desktop/server* e rinominalo semplicemente come `CanaryMod.jar`.

Puoi farlo dalla shell con il comando `mv` (*move file*, cioè "sposta file") e il carattere jolly (`*`), che sta per "tutti i numeri", così non dovrai digitarli tu:

```
$ cd server
$ pwd
/Users/andy/Desktop/server
$ ls
CanaryMod-1.7.10-1.1.0.jar
$ mv CanaryMod*.jar CanaryMod.jar
$ ls
CanaryMod.jar
```

Il codice di esempio del libro contiene la sottocartella *runtime*, dove vedrai uno script di avvio chiamato `start_minecraft`. Copialo nella tua cartella del server usando il comando `cp`. Sei ancora in questa cartella, quindi, puoi copiare usando semplicemente `..` per riferirti al Desktop e . per riferirti alla cartella corrente *server*:

```
$ cp ../code/runtime/start_minecraft .
```

Utilizzeremo lo script `start_minecraft` per avviare ed eseguire il server. Eseguirà il comando Java, passando un'opzione per garantirti che avrai abbastanza memoria per lavorare (la magica `-Xmx1024M`) e poi passando il file `CanaryMod.jar` per l'esecuzione:

```
java -Xmx1024M -jar CanaryMod.jar
```

La prima volta che il server di Minecraft si mette al lavoro, crea una serie di file, tra cui quello predefinito `World`, e poi esce. Avvialo (sempre nella cartella *server*):

```
$ ./start_minecraft
```

Nota che ho usato `./` come parte del nome del comando, così da avviarlo dalla cartella corrente. Se questa (.) è nel tuo percorso, non ti servirà usare la sequenza

`./`.

Se ti appare un messaggio di errore che dice `./start_minecraft: Permission denied`, dovrai digitare la riga seguente per rendere il file eseguibile:

```
$ chmod +x start_minecraft
```

Una volta che il server è avviato, Minecraft genera del testo sul tuo terminale. Il codice seguente ti mostra come appare sul mio computer (i nomi delle tue cartelle, le indicazioni dell'ora e i tuoi numeri di versione potrebbero essere diversi):

```
$ cd Desktop
$ cd server
$ ./start_minecraft
Please wait while the libraries initialize...
Starting: CanaryMod 1.7.10-1.1.0
Registered xml Database
Found 24 plugins; total: 24
[10:48:32] [CanaryMod] [INFO]: Starting: CanaryMod 1.7.10-1.1.0
. . .
[INFO]: You need to agree to the EULA in order to run the server.
Go to eula.txt for more info.
...
```

In realtà il testo generato è molto di più e qui ne ho tagliato un po', ma l'idea è questa. Da qualche parte nel testo si parla di accettare i termini del contratto di licenza (*license agreement*). Per farlo, dovrai modificare il file `eula.txt` appena creato nella cartella *server*. Se accetti gli accordi EULA (*End-User License Agreement*) di Mojang, modifica la riga che contiene `eula=false` in `eula=true` e salva il file. Ora puoi riavviare il server con `./start_minecraft`. Verrà generato dell'altro testo, ma questa volta vedrai il prompt `>`, in attesa della tua digitazione.

Ancora una cosa prima di iniziare: dovrai assegnarti i privilegi di operatore (*op*). Per farlo, digita il comando `op` con il tuo nome utente di Minecraft al prompt del server (alcune delle vecchie versioni di Canary mostrano un messaggio di errore al comando `op` anche se questo funziona; ignorali):

```
> op AndyHunt
[14:32:36] [CanaryMod] [INFO]: [SERVER] Opped AndyHunt
>
```

ERRORI ALL'AVVIO

In alcune versioni di CanaryMod, all'avvio, potresti vedere un messaggio di errore come `Error on line 1: Premature end of file. net.canarymod.database.exceptions.DatabaseWriteException`. Non spaventarti: in realtà vuole solo dire che il server ha "inciampato" mentre cercava di impostare il tutto. Elimina i file in *Desktop/server/db* e riprova.

Senza i privilegi di operatore, non potrai distruggere i blocchi né inserire nulla

nel gioco, quindi non saltare questo passaggio!

Se hai intenzione di invitare i tuoi amici a giocare sul tuo server, puoi rendere anche loro degli operatori oppure puoi dare a tutti i visitatori il permesso di costruire:

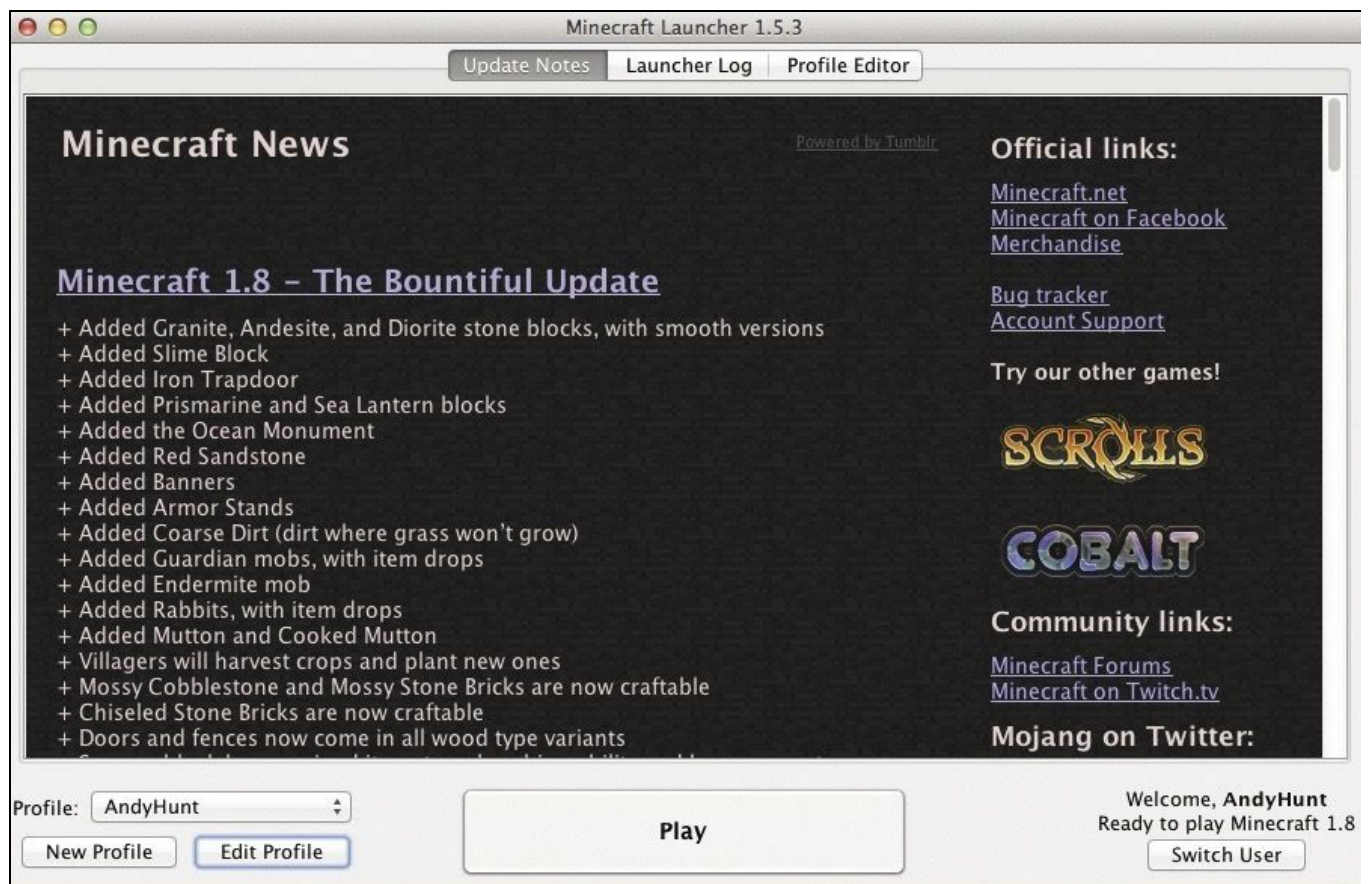
```
> groupmod permission add visitors canary.world.build
> [14:33:39] [CanaryMod] [INFO] [MESSAGE]: Permission added
```

Quando sei pronto, puoi uscire quando vuoi dal server di Minecraft digitando il comando `stop`, così:

```
>stop
[10:50:10] [CanaryMod] [INFO] [NOTICE]: Console issued a manual shutdown
[10:50:10] [net.minecraft.server.MinecraftServer] [INFO]: Stopping server
...
[10:50:10] [CanaryMod] [INFO]: Disabling Plugins ...
$
```

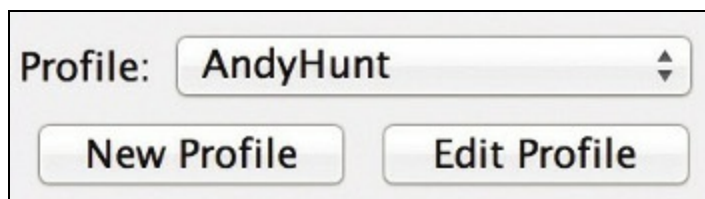
Verrai riportato al prompt della riga di comando.

Riavvia il server e lascialo in esecuzione, così proveremo a connetterci dal client.

Ora avvia il client di Minecraft e connettiti usando il tuo nome utente e la tua password di Minecraft. Con un po' di fortuna, vedrai un *launcher*.

Nota il numero della versione nell'angolo in basso a destra: qui ti dice che si può giocare con un server nella versione 1.8.

Quando avvii il client di Minecraft, puoi dirgli quale versione usare. Questa dovrà coincidere con la versione di Canary che hai scaricato.

Quando ho scritto questo libro, la versione più recente del server di Canary era la 1.7.10, ma il client che hai appena installato è Minecraft 1.8, che CanaryMod non supporta, e che quindi non funzionerà. Devi dire al client di Minecraft di utilizzare la versione corretta.

Nel programma di avvio di Minecraft, fai clic su *Edit Profile* in basso a sinistra.

Nella finestra di dialogo che si apre, cerca l'impostazione *Use Version*. Modifica questa opzione in modo da utilizzare la versione corrente del tuo server di Minecraft e fai clic su *Save Profile*.

Quando leggerai questo libro, i numeri delle versioni potrebbero essere cambiati completamente. L'importante è controllare sempre che la versione del client coincida con quella del server. Al termine, premi il pulsante *Play*.

Ora fai clic *Multiplayer*. Adesso devi aggiungere il tuo server locale. Fai clic su *Add Server* e digita un nome per il server. Usa `localhost` per l'indirizzo del server, come qui mostrato:

Il tuo server comparirà nell'elenco (insieme a tutti gli altri a cui ti connetti di solito), come vedi qui:

Scegli il tuo server dall'elenco e fai clic su *Join Server*. Benvenuto!

Se per qualche motivo la connessione si interrompe, potrebbe essere perché non c'è corrispondenza tra versioni. Ricontrolla la versione del client che stai usando.

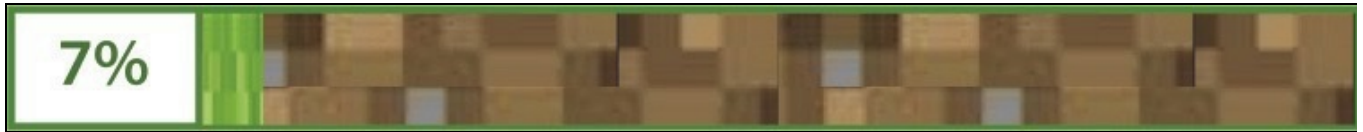
Una volta corretto il problema, entrerai nel mondo di Minecraft sul tuo server locale del gioco. Congratulazioni!

Puoi approfittarne per porre qualche base nel tuo mondo di Minecraft, per esempio per costruirti una casa prima che arrivino i Creeper…

Per continuare

Abbiamo iniziato bene. Ora hai un server di Minecraft completo e operativo sul tuo computer.

Nel prossimo capitolo ci rimboccheremo le maniche e compileremo e installeremo il nostro primo, vero plug-in.

La tua toolbox

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.

Creare e installare un plug-in

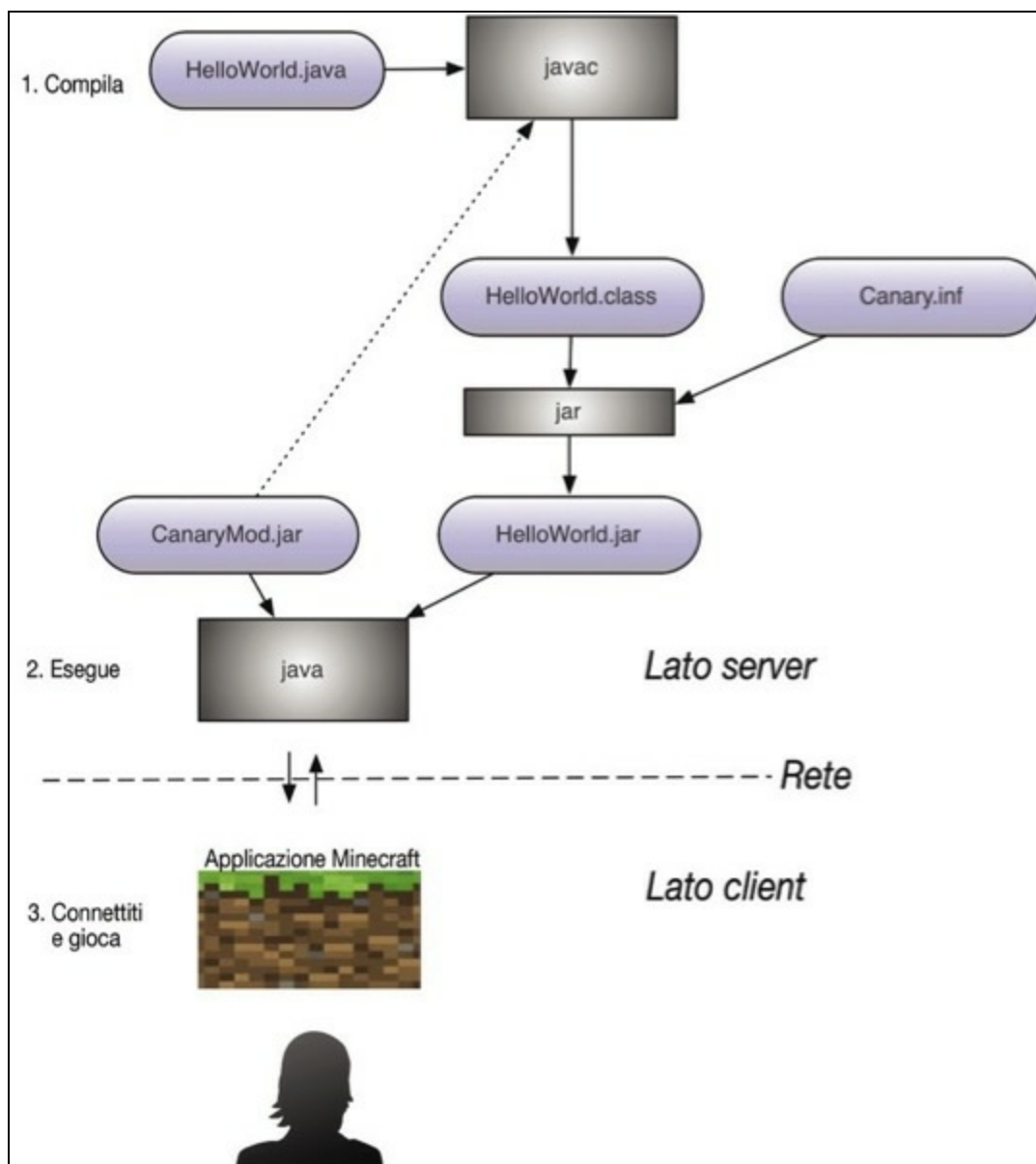
Ora che abbiamo installato tutti gli strumenti di base, possiamo costruire il primo plug-in. Non farà granché, ma ti permetterà di imparare a creare ed eseguire altri plug-in, e funzionerà come punto di partenza (uno scheletro) per tutti quelli che scriveremo nel resto del libro.

Come succede che le istruzioni che digiti vengono eseguite su un server di Minecraft? Vediamo l'intero processo.

A te spetta digitare le istruzioni in linguaggio Java (il codice sorgente) e salvarle in un file di testo, dopodiché il compilatore Java, javac, legge questo file lo converte in qualcosa di eseguibile dal computer. Hai già visto questo processo con il programma `CreeperTest.java` che hai scritto in precedenza.

Per il codice sorgente che digiti in un file chiamato `CreeperTest.java` ottieni un file *binario* (non di testo) chiamato `CreeperTest.class`. Un file binario non è altro che un file di numeri, che a noi umani dice poco, ma che il computer riesce a leggere.

Visto che un programma in genere finisce per utilizzare moltissimi file `.class`, di solito questi vengono salvati in un file JAR (`.jar`), da cui Java esegue il codice. Java (inteso come il programma) legge i file `.class` e i file JAR per generare un processo di esecuzione sul computer. Nel caso di Minecraft, si tratta del processo del server a cui i tuoi client del gioco si connetteranno. Per adesso l'unico client sarai tu.

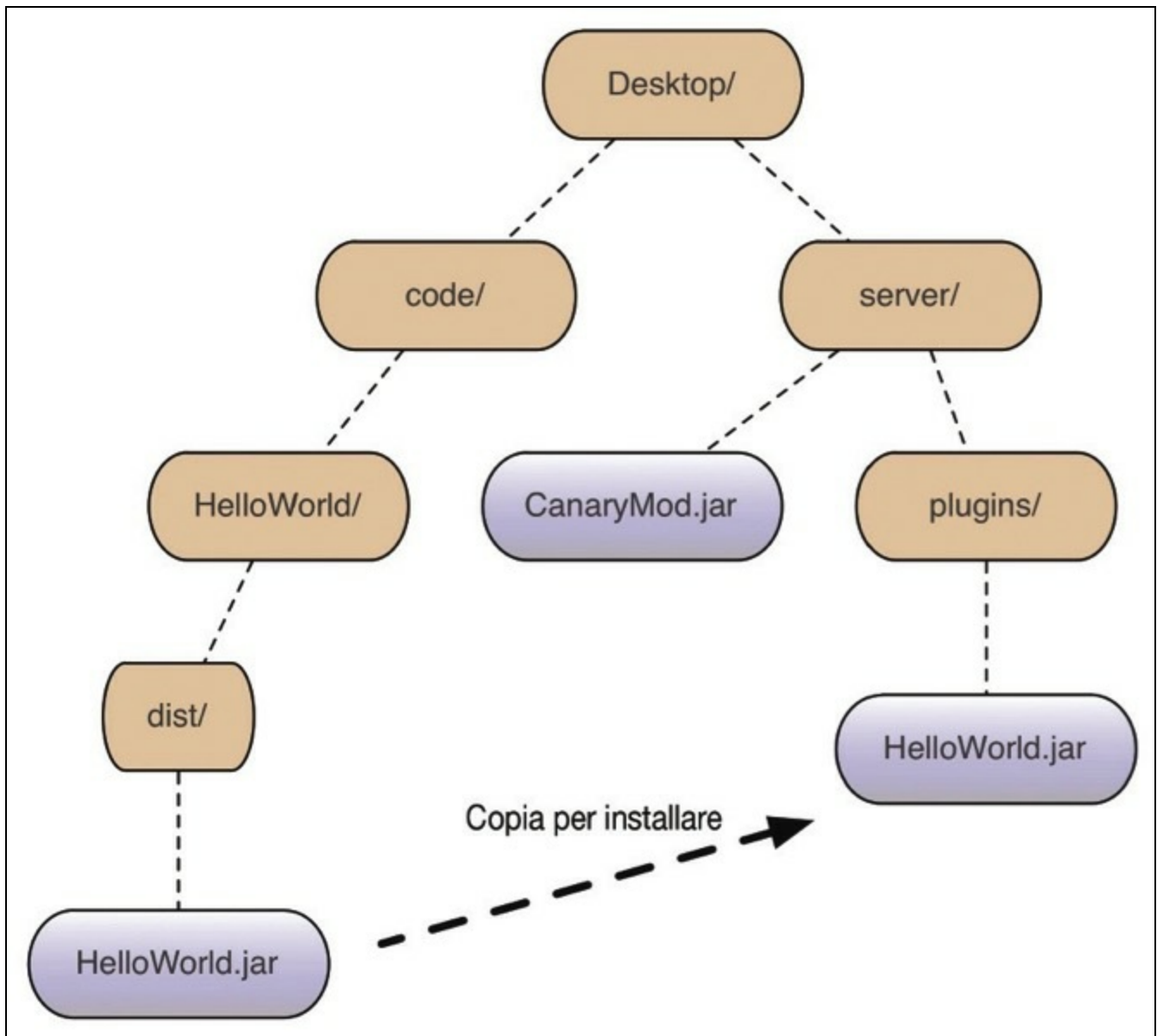
La prossima figura mostra come le varie parti vengono assemblate. Il compilatore javac prende il tuo codice sorgente Java e le definizioni che trova in `CanaryMod.jar` e produce un file `.class`. Questo file viene "confezionato" insieme al file `Canary.inf` in un JAR che è il tuo plug-in. Durante il runtime, Java avvia il server da `CanaryMod.jar` e carica il tuo plug-in dal suo JAR.

Nel mondo Java, devi collocare tutti questi file in alcuni punti specifici perché tutto funzioni. Abbiamo già creato una struttura di cartelle del genere in precedenza, pronta per la tua versione del plug-in `HelloWorld`. Per aiutarti, ho inserito un plug-in completo per te nella cartella *HelloWorld* nel codice del libro, che dovresti aver scaricato.

In *Desktop/code/HelloWorld*, troverai quindi un albero di cartelle del codice sorgente, sotto a *src*. Vedrai anche una cartella *bin* in cui vengono creati i file `.class` compilati e una cartella *dist* in cui i file `.class` e quelli di configurazione vengono riuniti in un file JAR. Quando sarai pronto per condividere il tuo plug-in con gli altri, darai loro il file JAR.

HelloWorld è una cartella di sviluppo. Probabilmente ne avrai una per ogni plug-in che svilupperai, e ciascuna avrà le proprie cartelle *src*, *bin*, *dist* e così via.

Nella tua cartella del Desktop (*Desktop/server*) hai i file del server di Minecraft, compreso `CanaryMod.jar`, che contiene tutti gli elementi che ti servono per eseguire il gioco e quelli che utilizzerai per sviluppare il codice nei mondi di Minecraft.

In *server*, esiste poi una cartella *plugins* che il server di Minecraft utilizzerà quando è in esecuzione, oltre che la cartella *lib* (la useremo alla fine del capitolo per la libreria `EZPlugin`).

Quando si lavora sul codice nella cartella di sviluppo, l'ultimo passo prima del test sul server è quello di copiare il file JAR nella cartella *plugin* del server (vedi la prossima figura).

Vedremo come farlo in automatico tra poco.

Java tende a utilizzare percorsi e file di configurazione per specificare dove risiedono tutti questi file e cartelle. A volte la cosa si rivela complessa, perché ci sono molti elementi che si spostano, ed è frustrante quando Java non riesce a trovare un file importante che hai proprio davanti a te: il fatto che *tu* sappia dove si trova, non significa che lo sappia anche Java.

Diamo ora un'occhiata al codice sorgente, e poi vediamo come costruirlo e installarlo.

Plug-in: HelloWorld

Nel mondo della programmazione, è tradizione partire con un semplice programma di prova che visualizza il testo “Hello, World” (“Ciao mondo”). Inizieremo quindi a costruire ed eseguire un plug-in esistente che visualizza un messaggio di saluto in Minecraft, anche se il nostro sarà un po’ diverso…

Di seguito vedi il codice sorgente Java del plug-in `HelloWorld`, che trovi già pronto nel file `~/Desktop/code/HelloWorld/src/helloworld/HelloWorld.java`. Contiene un sacco di roba misteriosa. (Se non hai ancora scaricato i file del libro sul Desktop, fallo adesso all’indirizzo http://media.pragprog.com/titles/ahmine2/code/ahmine2-code.zip. Puoi usare `unzip` dalla riga di comando per aprire l’archivio.)

HelloWorld/src/helloworld/HelloWorld.java

```
1 package helloworld;
2 import net.canarymod.plugin.Plugin;
  import net.canarymod.logger.Logman;
  import net.canarymod.Canary;
  import net.canarymod.commandsys.*;
  import net.canarymod.chat.MessageReceiver;

3 public class HelloWorld extends Plugin implements CommandListener {

    public static Logman logger;

 public HelloWorld() {
   logger = getLogman();
 }

 @Override
 public boolean enable() {
   logger.info(“Starting up”);
   try {
     Canary.commands().registerCommands(this,  this, false);
   } catch (CommandDependencyException e) {
     logger.error(“Duplicate command name”);
   }
   return true;
 }

 @Override
 public void disable() {
 }

4 @Command(aliases = { “hello” },
            description = “Displays the hello world message.”,
            permissions = { “” },
            toolTip = “/hello”)
5 public void helloCommand(MessageReceiver caller, String[] parameters)
   { String msg = “That’sss a very niccce EVERYTHING you have there...”;
     Canary.instance().getServer().broadcastMessage(msg);
   }
 }
```

Non scoraggiarti se ti sembra un discorso scritto in qualche lingua extra-terrestre o in elfico. Avrà più senso man mano che procediamo con i capitoli. Concentrati invece su quello che ti è familiare: ci sono alcune parole in inglese (abbastanza simili all'italiano, tra l'altro), come `import`, `public` e `return`, e quelle che potrebbero essere delle frasi o delle dichiarazioni, che terminano tutte con il punto e virgola (`;`). Ci sono poi alcuni caratteri insoliti come `{` e `}` che sembrano avere una qualche importanza.

Cosa vuol dire tutto questo? Questo plug-in implementa un comando da utente, `/hello`, che diffonde il tradizionale saluto dei Creeper, ovvero "`That'sss a very niccce EVERYTHING you have there...`" a tutti i giocatori di Minecraft che sono online.

Nota che al punto **3** il nome di questo plug-in viene dichiarato come `public class HelloWorld`: è lo stesso nome di quello del file che contiene il codice, `HelloWorld.java`. Questo segmento di codice è anche impostato come un *package* (un pacchetto), ossia un gruppo di file correlati, tramite lo stesso nome al punto **1**. Il nome del package è scritto tutto in minuscolo, e corrisponde al nome della cartella in cui si trovano i file del codice sorgente Java, sotto a *src*`/` in `helloworld/HelloWorld.java`. È importante che tutti i nomi nei vari punti coincidano: basta un refuso in uno per ottenere degli errori.

Le istruzioni `import` (le vedi al punto **2**) vanno usate per accedere ad altri elementi che potrebbero servirti nel plug-in, come parti della libreria di Canary e altre librerie Java. Se dimentichi di includere un'istruzione `import` per qualcosa che ti occorre, otterrai un errore "cannot find symbol" ("non trovo il simbolo"), perché Java non capisce cosa vuoi dirgli. Per tua comodità, nell'Appendice F ho incluso un elenco di tutto quello che importeremo.

Il codice del nostro plug-in inizia al punto **3**, mentre al punto **4** c'è un'annotazione `@Command` (una specie di tag, non un codice vero e proprio) che descrive il comando stesso. Il codice per quel comando inizia al punto **5**.

Vedremo tutto meglio più avanti; per adesso dobbiamo far sapere al server che abbiamo un plug-in pronto per essere caricato.

Configurare con Canary.inf

Questo codice sorgente da solo serve a poco; ti occorre anche un file di configurazione per far sì che Minecraft possa trovare e lanciare il plug-in. Il file di configurazione si chiama `Canary.inf`, e ha questo aspetto:

HelloWorld/Canary.inf

```
main-class = helloworld.HelloWorld name = HelloWorld
author = AndyHunt, Learn to Program with Minecraft
Plugins version = 1.0
```

Vediamo una descrizione di quello che serve al file. Per ora non preoccuparti troppo dei dettagli: avrà tutto più senso man mano che andremo avanti con il libro.

main-class

Nome del package e della classe che verranno eseguiti da Java per avviare il plug-in (*package.nomeclasse*).

name

Nome del plug-in, in questo caso `HelloWorld`.

author

Nome dell'autore (tu).

version

Numero della versione del tuo plug-in. Parti con un numero basso e incrementalo man mano che rilasci una nuova versione del plug-in agli altri.

Costruire e installare con build.sh

I comandi che digiti nella tua finestra del terminale possono anche essere salvati in un file; in questo modo puoi eseguirli più e più volte senza doverli ridigitare. Chiamiamo questo meccanismo *script della shell*, ed è un altro metodo di programmazione del computer.

La costruzione di un plug-in è solo leggermente più complessa della compilazione di un singolo file Java come quella del Capitolo 2; ciononostante, richiede molti comandi che non dovrai ridigitare ogni volta.

Per renderti le cose più facili, ho creato per te uno script chiamato `build.sh` che eseguirà i tre passi principali.

1. Usa `javac` per compilare il sorgente `.java` nei file `.class`.
2. Usa `jar` per archiviare i file `.class`, il file manifest e il file di configurazione.
3. Copia il file JAR sul server.

(Nel caso di progetti più grandi, qualcuno si serve di strumenti come Ant, Maven o Gradle nel momento in cui le operazioni diventano più complesse e vanno gestite le dipendenze tra molte parti. Ma è troppo per noi.)

Dopo tutto questo, devi fermare e poi riavviare il server così che possa applicare le modifiche. Lo script deve sapere dove si trova la tua cartella del server.

In cima allo script trovi questa riga:

```
MCSERVER=$HOME/Desktop/server
```

Nella maggior parte dei casi questo funziona se la cartella del server si trova sul Desktop. In caso contrario, dovrai modificare il file `build.sh` e cambiare la cartella in modo che `MCSERVER=` punti alla cartella del tuo server locale di Minecraft. Grazie all'impostazione `MCSERVER=` lo script capisce dove trovare il server.

Passa a *HelloWorld* come cartella corrente. Da qui, esegui lo script `build.sh` (è quello che faremo d'ora in avanti per ciascun plug-in):

```
$ cd
$ cd Desktop
$ cd code/HelloWorld
$ ./build.sh
```

(Ricorda: anche se accedi al Desktop in modo diverso, nel libro indicherò sempre l'operazione come `cd Desktop`. Potresti dover usare il comando `cd ~/Desktop` o

partire dalla tua cartella *home* e poi scendere nella struttura del file system; dipende da come è impostato il tuo sistema.)

Il risultato assomiglierà a questo:

```
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Verifica che il file sia stato davvero installato nella cartella del server:

```
$ cd Desktop
$ cd server/plugins
$ ls
HelloWorld.jar
```

Ehi, eccolo lì! C'è! Se dovessi ottenere qualche errore, prova a fare questi controlli.

- Se appare l'errore `./build.sh: Permission denied`, potresti dover digitare `chmod+x build.sh` per assegnare allo script un permesso di esecuzione.
- Se trovi errori di sintassi, riprendi la copia pulita dei file che hai scaricato dal sito web del libro, che non contiene modifiche.
- Se la compilazione ha successo ma ottieni un errore quando provi a copiare il file JAR, accertati che la cartella del server sia quella giusta.

Se lo script non riesce a trovare la cartella del server, modifica `build.sh` e poi il nome della cartella `MCSERVER=` indicando la posizione corretta del tuo server di Minecraft. (Controlla che sia scritta nel modo giusto e che inizi con un carattere `/`, così da avere il percorso completo per il server che parte fin dalla cartella root.)

Se ti trovi a dover compiere questa modifica, potresti dover fare lo stesso per tutti gli script di costruzione dei nuovi plug-in che incontrerai nel libro.

Si compila? Si installa? Eccellente! Ora hai un plug-in compilato pronto per essere utilizzato dal server di Minecraft.

Se il tuo server è ancora in esecuzione, non verrà a sapere di questo nuovo plug-in. Per informarlo, dovrai interromperlo e poi riavviarlo:

```
> stop
[14:36:24] [net.minecraft.server.MinecraftServer]  [INFO]: Stopping server
[14:36:24] [net.minecraft.server.MinecraftServer]  [INFO]: Saving players
[14:36:24] [net.minecraft.server.MinecraftServer]  [INFO]: Saving worlds
[14:36:24] [net.minecraft.server.MinecraftServer]  [INFO]: Saving chunks
            for level 'default'/Overworld
[14:36:24] [CanaryMod] [INFO]: Disabling Plugins ...
$ ./start_minecraft
08:47:32 [INFO] [HelloWorld] Loading HelloWorld v0.1
```

```
08:47:32 [INFO] [HelloWorld] Loaded.
08:47:32 [INFO] [HelloWorld] Enabling HelloWorld v0.1
08:47:32 [INFO] [HelloWorld] Starting up.
08:47:32 [INFO] Server permissions file permissions.yml is empty, ignoring it
08:47:32 [INFO] CONSOLE: Reload complete.
>
```

Ed ecco il messaggio di avvio del nostro nuovo plug-in `HelloWorld`. Se non vedi niente del genere, vuol dire che il server di Minecraft non riesce a trovare il plug-in. Assicurati che il file `HelloWorld.jar` sia nella cartella *plugins* del server, ferma il server e riavvialo.

Una volta che sei connesso ed entri nel mondo di Minecraft, puoi mettere alla prova il nuovo comando dalla finestra della chat del client. Dentro al gioco, inizia a digitare `/hello` e guarda cosa succede. Non appena scrivi il carattere `/`, ti accorgerai che stai digitando in una finestra di chat in fondo allo schermo. Premi il tasto Invio, e vedrai apparire il messaggio nella console del server e nella finestra del gioco. Questo è quello che appare nella console:

```
14:47:58 [INFO]: Command used by AndyHunt: /hello
14:47:58 [INFO]: That'sss a very niccce EVERYTHING you have there...
```

E questo è quello che appare nella finestra del gioco:

MUOVERSI IN MINECRAFT

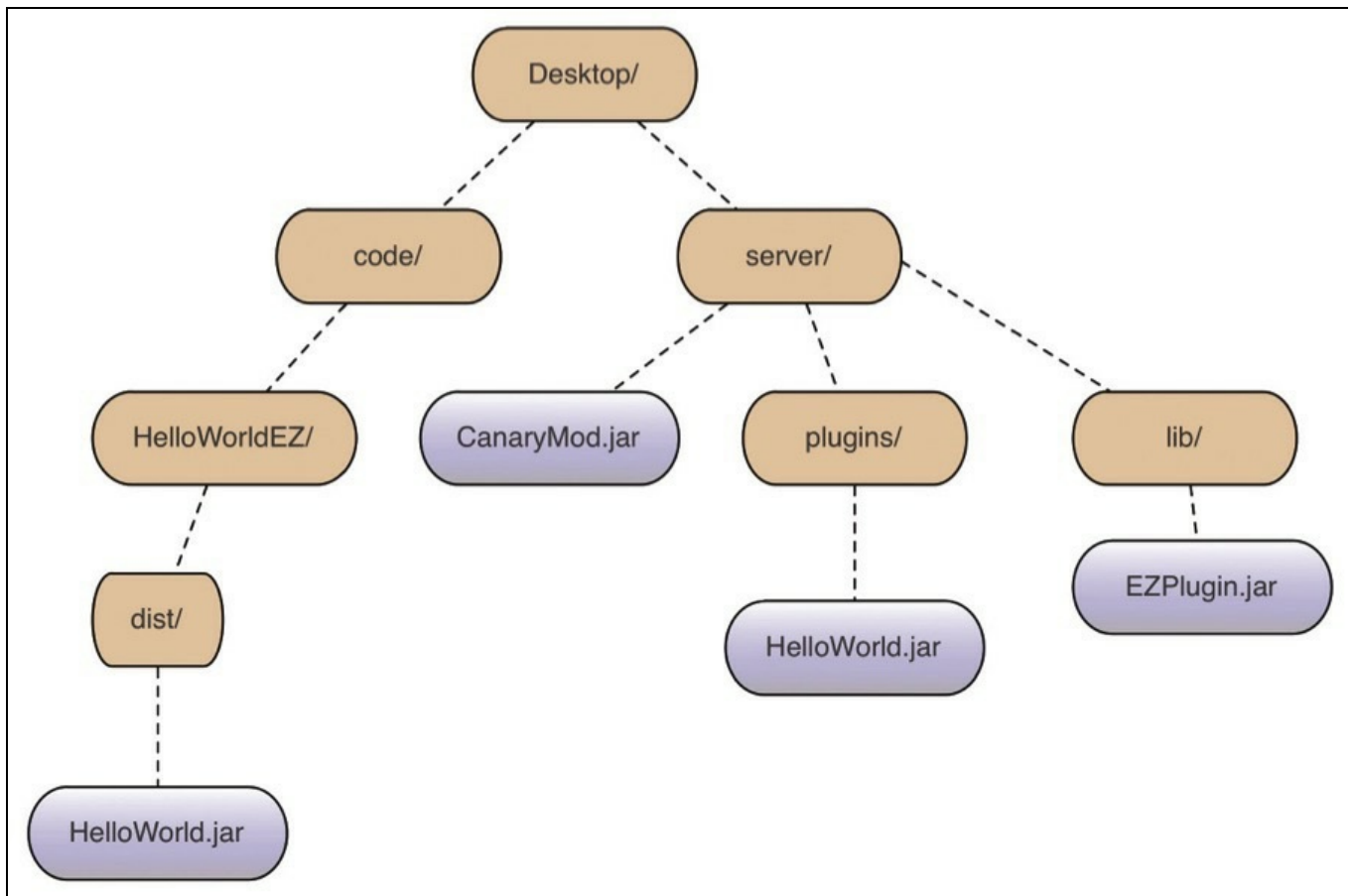
Se non hai molta confidenza con l'interfaccia grafica utente (GUI) di Minecraft, ti do alcuni

suggerimenti. Per saperne di più puoi anche guardare qualche tutorial su YouTube o leggere la documentazione ufficiale.
Con i tasti W, A , S e D ti sposti rispettivamente in avanti, a sinistra, all'indietro e a destra. Usa la barra spaziatrice per saltare. Usa il mouse per controllare la direzione davanti a te.
Per "colpire" le cose, premi il tasto sinistro del mouse, per esempio per colpire con una spada, una pala o affondare una piccozza.
Usa il tasto destro del mouse per manipolare le cose, per esempio per posare un oggetto o aprire una porta.
Fai precedere i comandi di Minecraft che digiti da uno slash (`/`). Puoi entrare in modalità Creativa con `/gamemode c` e poi tornare alla modalità Sopravvivenza con `/gamemode s`.

Usare EZPlugin

Man mano che proseguiremo con il libro, parleremo sempre più del linguaggio Java e spiegheremo cosa fanno tutti gli elementi che costituiscono `HelloWorld.java`. Tuttavia, metà di questo codice per adesso non è così importante, e soprattutto sarà sempre lo stesso per tutti i plug-in che utilizzeremo. Lo toglieremo di mezzo, così non dovrai continuare a guardarlo ogni volta che lavorerai con un plug-in.

Nel codice che hai scaricato ho incluso una libreria speciale chiamata `EZPlugin`. Ho spostato le cose che saranno simili per tutti i plug-in nel file `EZPlugin.java`. In questo modo, i prossimi plug-in saranno molto più piccoli e semplici da leggere.

Tutti gli altri plug-in (dopo `HelloWorld`) dipendono da `EZPlugin`, quindi dovremo costruire la libreria prima di poter continuare. Il processo di creazione la installerà nella cartella *lib* del server (*lib* sta per libreria).

Alla riga di comando, rendi la cartella *EZPlugin* quella corrente. Da qui, esegui lo script `build_lib.sh`. Nel mio caso, posso partire da qualsiasi punto e accedere alla mia cartella *home*, poi digitare `cd` fino alla cartella *Desktop*, arrivare a *code* e

infine a *EZPlugin*:

```
$ cd
$ cd Desktop
$ cd code/EZPlugin
$ ./build_lib.sh
```

Il risultato sarà più o meno questo:

```
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Controlla che il file sia stato effettivamente installato nella cartella *lib* del server:

```
$ cd Desktop
$ cd server/lib
$ ls
EZPlugin.jar
```

Una volta che `EZPlugin.jar` è in questa cartella, non serve più eseguire `build_lib.sh`; d'ora in avanti tutti i plug-in potranno utilizzarlo.

Adesso disfiamoci della prima versione di `HelloWorld.jar` nella cartella *plugins* del server:

```
$ cd ../plugins
$ ls
HelloWorld.jar
$ rm HelloWorld.jar
$ ls
$
```

Diamo un'occhiata a una versione più semplice di `HelloWorld` che utilizza `EZPlugin` e si trova in `code/HelloWorldEZ/src/helloworld/HelloWorld.java`.

HelloWorldEZ/src/helloworld/HelloWorld.java

```
package helloworld;
import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import com.pragprog.ahmine.ez.EZPlugin;

public class HelloWorld extends EZPlugin {
  @Command(aliases = { "hello" },
           description = "Displays the hello world message.",
           permissions = { "" },
           toolTip = "/hello")
  public void helloCommand(MessageReceiver caller, String[] parameters) {
    String msg = "That'sss a very niccce EVERYTHING you have there...";
    Canary.instance().getServer().broadcastMessage(msg);
  }
}
```

Nota che in alto ci sono alcune istruzioni `import`, mentre verso il basso c'è tutto

quanto riguarda i comandi, la parte che ci interessa di più. Questo sarà lo scheletro dei nostri prossimi plug-in.

Vai avanti, verifica di poter iniziare a costruire e che la libreria `EZPlugin` sia disponibile:

```
$ cd Desktop
$ cd code/HelloWorldEZ
$ ./build.sh
```

L'output sarà questo:

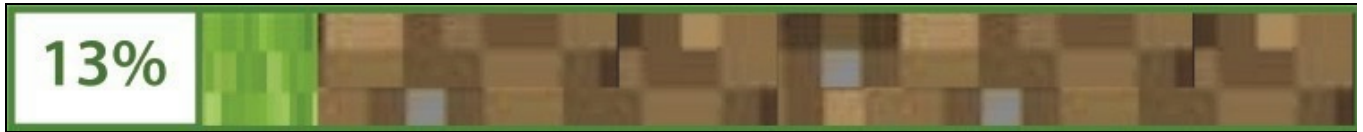
```
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Se non hai avuto successo, ricontrolla che `EZPlugin.jar` si trovi in *Desktop/server/lib*, rientra nel file `EZPlugin.jar` e ricostruiscilo, se necessario.

Per continuare

Complimenti! Hai appena compilato e installato un plug-in dal codice sorgente, l'hai installato su tuo server locale, ti sei connesso e l'hai testato. Hai anche costruito e poi installato la libreria `EZPlugin`.

Nei prossimi capitoli daremo un'occhiata un po' più approfondita al codice sorgente che costituisce un plug-in, e vedremo come lavora Java per permetterti di creare i tuoi plug-in.

La tua toolbox

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.

I plug-in hanno variabili, funzioni e parole chiave

Il codice sorgente del plug-in contiene molto testo che non ti ho ancora spiegato. Approfondiamo Java e diamo un'occhiata a cosa significa quella parte, perché è la materia prima della programmazione dei plug-in.

Come hai visto nel Capitolo 3, in Java un programma non è altro che un file di testo. Guarda `HelloWorld.java`: contiene diverse parole simili all'inglese e una punteggiatura strana.

Ogni cosa ha un significato, e inoltre il compilatore Java è piuttosto pignolo quando si tratta di leggere un testo. Tanto per cominciare, le parole vanno scritte secondo certe regole. `Entity` non è uguale a `Entitee`, così come `Player` non è uguale a `player`, e questo ti dimostra che contano anche le maiuscole e le minuscole: a volte devi usare le prime, a volte le seconde.

Tutti quei bizzarri caratteri che vedi hanno un significato speciale, e sono utilizzati per operazioni diverse. Ecco alcuni esempi (per adesso non preoccupiamoci dei dettagli).

`//`	Le doppie barre indicano un commento, che usi per lasciare qualche indicazione agli altri o come promemoria per te. Java ignora tutto quello che digiti dopo questo segno fino al termine della riga.

`/*`
`commento`
`più lungo`
`*/`
	Questa forma viene usata per i commenti più lunghi, sia su una riga sia più righe.

`( )`	Le parentesi tonde vengono usate quando si chiama una funzione e le si passano dei dati, come in `System.out.println("Hello, Creeper");`.
`[ ]`	Le parentesi quadre vengono usate quando si deve scegliere un elemento da un elenco, come in `second = myArray[1];`.
`{ }`	Le parentesi graffe vengono usate per indicare l'inizio e la fine di una sezione del codice.
`.`	Il punto viene usato per selezionare una parte di qualcosa, in genere una funzione che fa parte di un oggetto, come in `System.out.println()` o `player.getLocation()`.
`;`	Il punto e virgola viene usato per indicare la fine di un'istruzione nel codice. Dimenticarsi di digitarlo al termine di una riga di codice vuol dire irritare Java, che

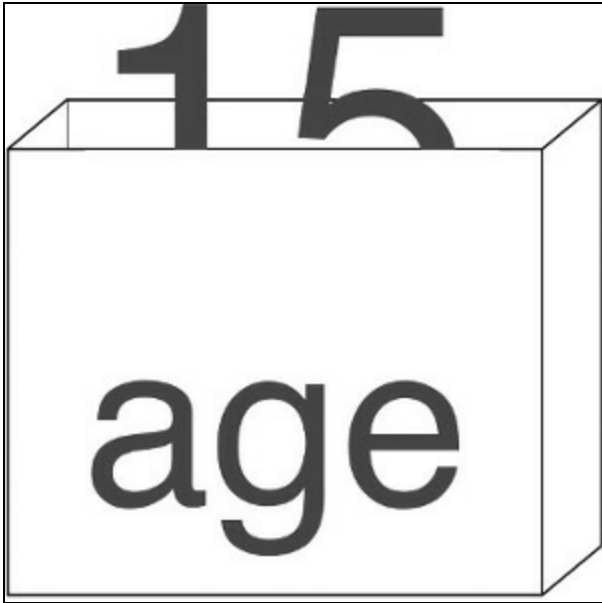
	generà centinaia di messaggi di errore.

Alcune delle parole che scrivi sono speciali, e devi usarle così come ti chiede Java. Su altre hai più libertà, e possono andare da `GiocatoreImmortale` a `testaditopo`.

Quando scrivi un programma o un plug-in in Java, utilizzi tutti questi pezzettini di testo per creare due tipi di elementi: dati e istruzioni. Questo vale per qualsiasi programma per computer così come per un plug-in di Minecraft. È tutta una questione di dati e di istruzioni. Vediamo come lavorare con questi elementi.

Per prima cosa ti mostrerò come dichiararli e utilizzarli; poi dovrai provarli nel tuo primo plug-in.

Tenere traccia dei dati con le variabili

I dati sono fatti, cose come la tua età o il colore della tua bicicletta. Per lavorare con i fatti in Java, devi conservarli in *variabili*. Una variabile è un contenitore di dati, una specie di scatola in cui puoi inserire le cose.

In Minecraft, usiamo le variabili per tenere traccia di informazioni come chi sono i partecipanti, qual è lo stato di salute o la posizione di un giocatore e altri dati che ci occorre sapere. (Tra poco aggiungerai alcune variabili a un segmento di codice, un plug-in per costruire una casa.)

Una variabile è la scatola che contiene questi dati. Volendo puoi anche metterle un'etichetta, ma solo per tua comodità, senza che questo influisca su quanto racchiude.

In questo esempio creeremo una variabile chiamata `age` (età) e diremo a Java di impostare il suo valore a 15, prima in due passi:

```
int age;
age = 15;
```

e poi in uno:

```
int age = 15;
```

La cosa importante qui è che stai comunicando a Java il *tipo* di variabile che vuoi creare, in questo caso un numero intero (senza decimali). Un numero intero è chiamato *integer*, in Java abbreviato in `int`.

Oltre a `int` esistono numerosi altri tipi, compresi `float` e `double`, che contengono

decimali, o `String`, che contiene una sequenza di caratteri, come `"Hello, World!"`. Anche tu puoi creare dei tipi, come `NumTiri` o `CampanacciMucche`, e lo stesso Minecraft ne definisce di suoi, come `Player` o `Location`. Ogni volta che dichiari una variabile in Java, devi specificare di che tipo è. Java non è in grado di capirlo da solo, devi dirglielo tu.

Puoi creare la variabile e assegnarle un valore in un unico passo, oppure puoi prima creare la variabile e poi darle un valore. In entrambi i casi, puoi inserire nella variabile un nuovo valore ogni volta che vuoi:

```
int age = 15;
age = 39;
age = 21;
```

Tuttavia, non puoi dichiararla una seconda volta. Questo non funzionerà:

```
int age = 15;
int age = 21;
// Errore!
```

Se ti occorre, dichiarala una volta come `int age` e poi usa `age =` per cambiare il valore.

Ora, il fatto che tu abbia "etichettato la scatola" come *age* non significa che questa debba contenere per forza delle età. Niente ti impedisce di inserire altri numeri:

```
int age = 2048;
```

Questo è un codice Java perfettamente valido, perché 2048 è un numero accettabile. Potrebbe indicare quanti anni ha una rovina storica, ma non è un numero plausibile per l'età di una persona. Impostarla come l'età di qualcuno sarebbe stupido, ma nessuna legge impedisce la stupidità… Sta quindi a te dare alle variabili dei nomi che abbiano un senso. Per esempio, potrebbe essere meglio rinominare la variabile come `ageInYears` (età in anni), per evitare problemi come quelli che portarono al fallimento nella missione Mars Climate Orbiter, in cui un gruppo di programmatori della sonda usò il sistema metrico decimale e un altro le unità di misura imperiali (http://it.wikipedia.org/wiki/Mars_Climate_Orbiter). Ops.

Java prova a sempre a impedirti di mescolare cose che non c'entrano tra loro. Anche se non sa che usi un `int` come "età di una persona", conosce la differenza tra un `int`, un `float`, una `String` e così via, e non ti permette di mischiare i vari tipi. Se hai dichiarato che una variabile deve essere un integer, non puoi memorizzare una `String` al suo interno, altrimenti, si genera un errore:

```
int age = "Old enough to know better";  // Errore!
```

Qui abbiamo provato a memorizzare una stringa di caratteri in un `int`. Non va bene, e Java protesta. Non puoi nemmeno fare il contrario, per esempio memorizzare un valore intero (`42`) in una variabile dichiarata come `String`:

```
String answer = 42;   // Errore!
```

Tuttavia, se necessario e se ha senso, nel caso di tipi comuni di numeri e stringhe, puoi effettivamente convertire i valori in una direzione o nell'altra. Per esempio, immagina di leggere un valore numerico da qualcosa digitato dall'utente che ti è stato passato come una `String` chiamata `str`. Puoi renderlo un `int` così:

```
String str = "1066";
int value = Integer.parseInt(str);
// il valore ora è impostato a 1066 come numero
```

La conversione è quindi possibile, ma devi farla da te, perché Java non può intuirla. L'Appendice E contiene un elenco delle conversioni di tipo più comuni, comprese alcune che non tratteremo.

Continuiamo.

Plug-in: BuildAHouse

Ho già preparato un plug-in per te; tutto quello che dovrai fare è dichiarare alcune variabili, dopodiché potrai dare il comando `/buildahouse`.

Per prima cosa, accedi al plug-in `BuildAHouse` nel codice del libro che hai scaricato:

```
$ cd Desktop
$ cd code/BuildAHouse/src/buildahouse
$ ls
BuildAHouse.java MyHouse.java
```

Modificherai il file `MyHouse.java`, che è una piccola parte del plug-in (non guardare il resto, ancora!). Ora come ora appare così:

BuildAHouse/src/buildahouse/MyHouse.java

```
package buildahouse;
public class MyHouse {
  public static void build_me() {
    // Dichiara la larghezza
    // Imposta il numero di blocchi per la larghezza
    // Dichiara l'altezza
    // Imposta il numero di blocchi per l'altezza

    BuildAHouse.buildMyHouse(width, height);
  }
}
```

Se provi a compilarlo e a installarlo con `./build.sh` come hai fatto con `HelloWorld`, ottieni due errori:

```
$ cd Desktop
$ cd code/BuildAHouse
$ ./build.sh

Compiling with javac...
src/buildahouse/MyHouse.java:10: cannot find symbol
symbol  : variable width
location: class buildahouse.MyHouse
    BuildAHouse.buildMyHouse(width, height);
                             ^
src/buildahouse/MyHouse.java:10: cannot find symbol
symbol  : variable height
location: class buildahouse.MyHouse
    BuildAHouse.buildMyHouse(width, height);
                                    ^
2 errors
```

Questa è la tua prima missione: dichiarare e impostare una variabile `int` chiamata `width` (larghezza) e una variabile `int` chiamata `height` (altezza), impostandole a qualcosa di accettabile per una casa, che magari non dovrà essere più bassa e più stretta di 5×5 blocchi, o di 10×10 blocchi se vuoi più spazio.

Cancella le righe dei commenti e sostituiscile con le tue due variabili `width` ed

`height`. Salva il file ed esegui nuovamente `build.sh`:

```
$ ./build.sh
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Interrompi il server e poi riavvialo, quindi connetti (o riconnetti) il tuo client di Minecraft. Scegli una bella posizione nel paesaggio dove costruire la tua casa e digita il comando `/buildahouse`.

Bam! Eccoti nella tua nuova casa a prova di Creeper, che è stata costruita esattamente con le dimensioni da specificate attraverso le variabili `width` ed `height`. Fai clic con il tasto destro del mouse per aprire la porta.

Numeri di tipo diverso

Java distingue tra numeri interi (*integer*) senza decimali, come 8, e numeri decimali o a virgola mobile (*floating-point*), come 10.125.

NOTA

Java usa la notazione inglese per i decimali, cioè scrive il punto al posto della virgola.

Come già visto, per gli interi si usa una variabile `int`. I numeri decimali possono

essere `float` o `double`. Un `double` è più grande e può memorizzare i numeri con più precisione, anche se per gestirli servono più spazio e potenza. I computer moderni sono però ormai piuttosto veloci e capienti, quindi quasi tutti i programmatori tendono a usare i `double` ogni volta che serve loro un numero decimale a precisione doppia.

Quando digiti un numero che ha una parte decimale, Java dà per scontato che tua abbia immesso un `double`:

```
3.1415 // Java lo legge come un double
```

Se ti serve che un numero sia `float`, devi aggiungere la lettera `f` al numero:

```
3.1415f // Ora è un float, non un double
```

Tra poco utilizzeremo dei `float` per riprodurre un effetto sonoro, perché è quello che richiede da Canary per impostare il volume e avviare l'audio. (Con la pratica, vedrai che userai i `float` piuttosto di frequente.)

Per esempio, consideriamo un semplice problema di divisione. In Java, la divisione si scrive con il carattere `/` (invece di `÷`), quindi per dividere il numero 5 a metà devi scrivere `5 / 2`. A seconda di come fai la divisione, però, il risultato potrebbe essere sorprendente.

- 5 / 2 dà 2 (solo 2, nient'altro).
- Ma 5 / 2.0 dà 2.5, come dovrebbe essere.
- Anche 5.0 / 2.0 dà 2.5.

Perché 5 diviso 2 dà solo 2? In un esercizio matematico, questo sarebbe considerato un errore. Qui però non stiamo dividendo dei numeri reali, ma dei numeri `int`, quindi il risultato è un altro numero intero, cioè un un altro `int`, senza decimali.

Se vuoi che la risposta includa i decimali, almeno uno dei numeri coinvolti deve essere decimale. Ecco perché 5 diviso 2.0 (nota .0) dà il risultato corretto 2.5. Qui si usa un `int` (5) e un `double` (2.0).

A volte non è necessario badare a decimali o resti. Se calcoli qualcosa che non ha parti decimali, allora va bene il metodo con tutti `int`. Dire "metà di un `Player`" (cioè di un giocatore) o "metà di una `Cow`" (cioè di una mucca) non ha senso (a meno che non tu stia preparando una bistecca). Se invece ti occorrono i decimali, almeno uno dei due numeri deve essere un `double` o un `float`.

Ecco un pratico elenco di alcuni degli operatori matematici più comuni che potrebbero servirti:

- Addizione: `+`
- Moltiplicazione: `*`
- Sottrazione: `-`
- Divisione: `/`

Funzionano esattamente come ti aspetti, ma ci sono delle scorciatoie:

```
int health = 50;
health = health + 10;
```

è uguale a

```
int health = 50;
health += 10; // È la stessa cosa
```

Entrambi i risultati in `health` vengono impostati a 60.

Puoi utilizzare espressioni come `+=` e `-=` per modificare il valore di una variabile senza doverne ripetere il nome. Un bel risparmio (i linguaggi di programmazione migliori permettono di digitare pochissimi caratteri e ottenere comunque qualcosa che abbia un senso).

Se devi solo aggiungere o sottrarre 1, c'è un metodo ancora più semplice per scriverlo:

```
int health = 50;

health--; // Sottrae 1 da health
health++; // Aggiunge 1 a health
```

Se vuoi usare della matematica più avanzata, includendo funzioni come seni e coseni, costanti come il pi greco e altre cose del genere, Java ha librerie specifiche pronte all'uso.

Stringhe di caratteri

Al mondo, però, non ci sono solo i numeri. Magari lo Stato o la tua scuola ti ha identificato con un numero come #132-54-7843, ma i tuoi amici ti chiamano per nome, che è una stringa di caratteri, come "Paolo" o "Ilaria".

In Java, le stringhe di caratteri si gestiscono con il tipo di variabile `String`. In Minecraft ne useremo parecchie, per i nomi dei giocatori, dei file, i messaggi e per qualsiasi altro testo di cui puoi cambiare il valore, come accade per i numeri.

Per specificare una stringa nel codice, devi racchiuderla tra doppie virgolette, `"in questo modo"`. Useremo le stringhe nei nostri plug-in e vedremo cosa farne man mano che procederemo.

Per concatenare le stringhe devi usare il segno più (`+`):

```
String first = "Jack";
String middle = "L.";
String last = "Squartatore";

String name = first + " " + middle + " " + last;
// Ora il nome sarà "Jack L. Squartatore"
```

Nota che le stringhe sono i tuoi *dati*, e sono diverse dai caratteri che usiamo per impartire delle istruzioni a Java (il nostro *codice* di programma).

Prova da solo

È venuto il momento di mettere in pratica tutti questi concetti: creeremo un semplice plug-in da zero.

Il plug-in `BuildAHouse` conteneva del codice che non ti ho mostrato che permetteva di costruire effettivamente la casa; tutto quello che hai dovuto fare è stato dichiarare alcune variabili. Adesso invece realizzerai un plug-in intero tutto da solo.

Per restare sul semplice, inizieremo con il visualizzare alcuni valori; chiameremo questo plug-in `Simple`.

Innanzitutto devi creare una cartella *Simple* per il nuovo plug-in, le due sottocartelle *src/* e *src/simple* e i file `Canary.inf`, `Manifest.txt` e `build.sh`, esattamente come hai visto in `HelloWorldEZ`.

Ho già creato la cartella *Simple* per te in *Desktop/code*, ma cerca di non guardarla, a meno che non resti bloccato. Prova a crearla da solo.

Visto che è un'operazione che dovrai svolgere spesso, ho preparato uno script della shell chiamato `mkplugin.sh` che ti darà una mano.

Nella tua cartella *Desktop*, esegui `mkplugin.sh` con il nome del plug-in che vuoi creare; questo genererà le cartelle secondarie sotto a quella corrente e ti permetterà di partire con il codice Java:

```
$ cd Desktop
$ code/mkplugin.sh Simple
$ cd Simple
$ ls
```

```
Canary.inf  Manifest.txt bin  build.sh  dist  src
$ cd src
$ ls
simple/
$ cd simple
$ ls
Simple.java
```

Il file `Simple.java` in realtà non fa niente.

Dovrai aggiungere del codice tra `Inserisci il tuo codice dopo questa riga` e `… termina il tuo codice prima di questa riga`, come mostrato nel prossimo listato.

Apri il file `Desktop/Simple/src/simple/Simple.java` nel tuo editor e preparati a digitare.

Plug-in: Simple

Simple/src/simple/Simple.java

```
   package simple;
   import net.canarymod.plugin.Plugin;
   import net.canarymod.logger.Logman;
   import net.canarymod.Canary;
   import net.canarymod.commandsys.*;
   import net.canarymod.chat.MessageReceiver;
   import net.canarymod.api.entity.living.humanoid.Player;
 1 import com.pragprog.ahmine.ez.EZPlugin;

   public class Simple extends EZPlugin {

     @Command(aliases = { "simple" },
               description = "Displays simple variable assignments",
               permissions = { "" },
               toolTip = "/simple")
     public void simpleCommand(MessageReceiver caller, String[] parameters) {
       if (caller instanceof Player) {
         Player me = (Player)caller;
         // Inserisci il tuo codice dopo questa riga:
2
         // ... termina il tuo codice prima di questa riga.       }
     }
   }
```

Per adesso non preoccuparti di tutto quello che c'è intorno, ne parleremo più avanti. Inserisci il nuovo codice come mostrato, sarà sufficiente. Procediamo.

Per cominciare, aggiungi due istruzioni importanti in cima al file, dopo le righe `import` al punto **1**. Digita quanto segue:

```
import net.canarymod.api.world.effects.SoundEffect;
import net.canarymod.api.world.position.Location;
```

Dopo `// Inserisci il tuo codice dopo questa riga` al punto **2**, fai quanto segue.

1. Crea una variabile integer chiamata `myAge` e impostala con la tua età.
2. Crea un'altra variabile integer chiamata `twiceMyAge` e impostala con lo stesso valore di `myAge` moltiplicato per 2.
3. Crea una variabile float chiamata `volume` e impostala a `0.1`.
4. Crea una variabile float chiamata `pitch` e impostala a `1.0`.
5. Crea una variabile float (decimale) chiamata `dayOnIo` e impostala a `152853.5047`. È il numero della durata in secondi di un giorno su Io, una delle lune di Giove (un posto interessante: ha oltre 400 vulcani attivi. Per saperne di più vai all'indirizzo http://it.wikipedia.org/wiki/Io_(astronomia)).
6. Crea una variabile stringa chiamata `myName` e impostala con il tuo nome.

7. Visualizza tutti questi valori inviando un messaggio via chat al giocatore usando `me.chat(`*string msg*`)`. Per esempio, per vedere il tuo nome, digita:

```
me.chat("My name is " + myName);
```

Inizia ogni messaggio con una stringa (come `"My name is "`, cioè "mi chiamo") e il segno più, poi aggiungi la tua variabile. Non dimenticare di inserire il punto e virgola alla fine di ciascuna istruzione.

Infine, aggiungi le righe per riprodurre un effetto sonoro nella tua posizione, usando i valori di `float` che hai appena dichiarato:

```
Location loc = me.getLocation();
playSound(loc, SoundEffect.Type.GHAST_SCREAM, volume, pitch);
```

Salva il file ed esegui `build.sh` nella cartella *Simple*. (Se il tuo server è attivo, assicurati di interromperlo; alcuni sistemi operativi generano degli errori se provi a installare dei nuovi JAR mentre il server sta lavorando.)

```
$ cd Desktop
$ cd Simple
$ ./build.sh
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Interrompi il server e riavvialo, quindi connettiti con il client di Minecraft. Esegui il tuo nuovo comando, `/simple`, e stupisciti davanti ai numerosi messaggi che appaiono sulla console (e all'effetto audio!).

Ora che funziona, prova a cambiare i valori per `volume` e `pitch`. Alza il volume a `1.0f` per ottenere un urlo terrificante. Per `pitch` (il tono), prova un valore molto basso come `0.1f` per ottenere un ringhio, e uno più alto (`10.0f`?) per un grido che squarcia l'aria.

Una volta digitato il nuovo valore nel codice sorgente Java, non dimenticare di fare quanto segue.

1. Salva il file.
2. Compila e installa con `./build.sh`.
3. Interrompi il server e riavvialo.
4. Riconnettiti con il tuo client di Minecraft.
5. Digita `/simple` nel client e goditi suoni e immagini.

Ecco come appare a me:

(In realtà non ho 99 anni come si vede qui, ma non ti dirò la mia vera età, né il mio numero di conto corrente o altro…)

Se vuoi vedere il mio codice per avere qualche suggerimento, dai un’occhiata a `code/MySimple/src/mysimple/MySimple.java`.

Organizzare le istruzioni in una funzione

Ora che sai come memorizzare tutti i tipi di dati nelle variabili, devi imparare a scrivere quelle istruzioni che compiono azioni con i dati, come stampare i messaggi o lanciare mucche in fiamme in Minecraft.

Come hai già visto, puoi dire a Java di fare delle cose. In Java, le righe di codice (istruzioni) vengono racchiuse tra una coppia di parentesi graffe, `{` e `}`. A questa sezione di codice devi assegnare un nome, e le istruzioni che contiene verranno eseguite in ordine, una riga dopo l'altra. Chiamiamo questo meccanismo *funzione* (o *metodo*).

Perché occuparsi delle funzioni? Non basta avere un grosso elenco di istruzioni e impartirle? Beh, sì, potrebbe bastare, ma si potrebbe anche creare un bel po' di confusione.

Prova a immaginare un elenco di istruzioni e ingredienti per preparare una torta con la glassa.

- Amalgama e inforna
- Farina
- Burro
- Zucchero
- Latte
- Uova
- Vaniglia
- Cacao in polvere
- Zucchero a velo
- Burro
- Latte
- Mescola e distribuisci sulla torta

Quale parte di questo elenco riguarda la preparazione della torta e quale quella della glassa? Forse la glassa inizia dal cacao in polvere. O magari è la ricetta di una torta con la base di cioccolato e una glassa alla vaniglia. Difficile da capire. Potrebbe funzionare così com'è, ma capire se c'è qualcosa che non va potrebbe diventare un problema. Peggio ancora se ti trovi a dover fare delle correzioni.

Immagina di avere qualche parente bizzarro che vuole che la sua torta abbia una copertura all'albicocca e arancia invece di una glassa al cioccolato (l'ho detto che era bizzarro). In quale punto devi fare le modifiche alla ricetta?

Invece di avere un'unica lista, supponi di dividerla in due fasi, dove ogni fase elenca gli ingredienti e i passaggi per la preparazione della sola torta:

- `makeChocolateCake`
- `makeVanillaFrosting`

Così è molto più chiaro. Se c'è qualche problema con la torta, sai dove andare a cercare. Se vuoi preparare una glassa diversa, puoi cambiarla facilmente:

- `makeChocolateCake`
- `makeOrangeApricotGlaze`

Questo è quanto sta dietro l'idea di funzione. Le funzioni sono un modo per raccogliere le istruzioni e i dati in gruppi che abbiano un senso. Hanno poi un valore aggiunto: puoi usare la stessa funzione (l'elenco di istruzioni) anche con dati leggermente diversi. Per esempio, potresti avere una funzione chiamata `makeFrosting` e chiamarla con dei gusti (`flavor`) diversi:

```
makeFrosting(flavor)
    sugar
    butter
    mix in "flavor"
    spread on cake
```

Dopodiché puoi passarle altri dati:

```
makeFrosting(vanilla)
makeFrosting(chocolate)
```

Ecco perché usiamo le funzioni: per rendere più facili da leggere e capire i lunghi elenchi di istruzioni (il codice) e per riutilizzare quegli stessi elenchi con dati leggermente differenti.

In questo esempio ci siamo presi una piccola licenza. La nostra funzione `makeFrosting` non è codice Java vero e proprio. Quando scrivi un'idea che si presta a essere trasformata in codice ma che non è propriamente un linguaggio di programmazione, usi uno *pseudocodice*. I programmatori utilizzano uno pseudocodice così come i pittori fanno uno schizzo del quadro prima di iniziare a dipingerlo: li aiuta a farsi un'idea.

Ma torniamo a Minecraft.

Definire le funzioni in Java

Ogni pezzo di codice che scriviamo in Java finisce in una funzione: è così che lavora questo linguaggio di programmazione. Alcune funzioni le abbiamo già viste fin dal primo plug-in.

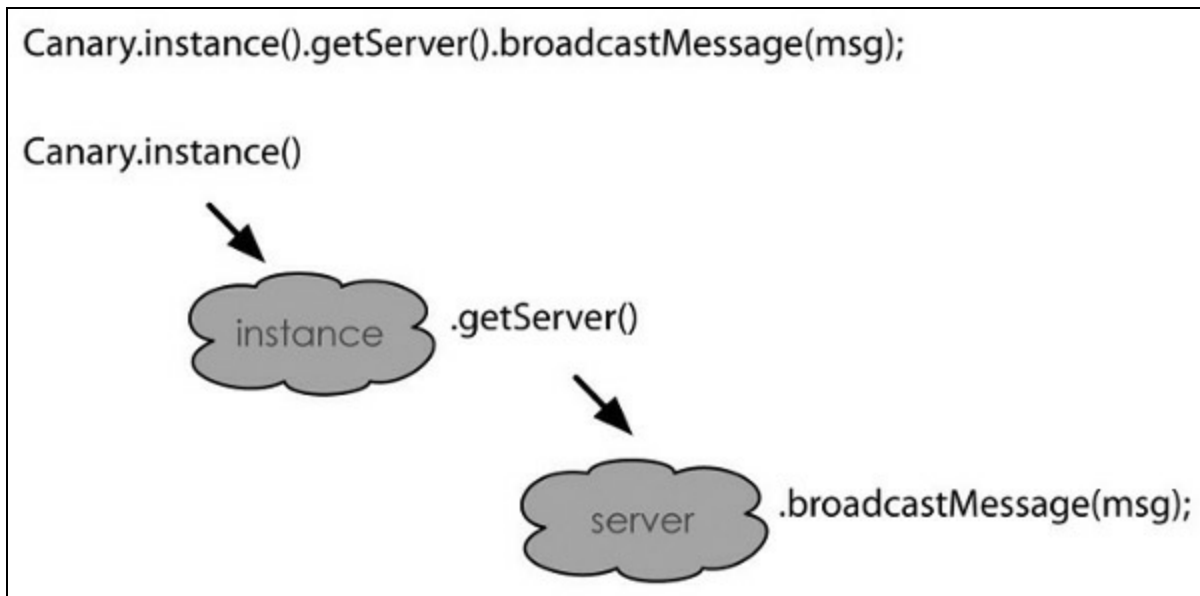
Tornando al plug-in `HelloWorld`, abbiamo dichiarato la funzione `helloCommand` che Minecraft chiama quando il gioco è in corso. Definiamo questo tipo di funzioni *entry point* (cioè “punti di inserimento”). Sono funzioni che il server di Minecraft chiama quando gli serve. Tu fornisci il codice, e il codice le chiamerà all’occorrenza.

In `helloCommand` chiamiamo altre funzioni. Ecco la sezione da `HelloWorld`:

```
public void helloCommand(MessageReceiver caller, String[] parameters) {
  String msg = “That’sss a very niccce EVERYTHING you have there...”;
  Canary.instance().getServer().broadcastMessage(msg);
}
```

Qui c’è una chiamata a `instance()`, una a `getServer()` e una a `broadcastMessage()`.

Java sa che stai chiamando una funzione grazie alle parentesi che si trovano dopo il nome; si aspetta che qualcuno abbia definito una funzione in base al nome e passerà il tuo messaggio alla funzione. Tutto quello che passi alle funzioni è un *argomento*. Quando gli argomenti sono inviati a una funzione, questa li riconosce come *parametri*. Si dice che i valori vengono *passati* o che la funzione viene *chiamata* con questi valori. Tutte le parole e le frasi si riferiscono allo stesso concetto.

Per esempio, la funzione `getServer()` non prende nessun argomento. Siccome usi le parentesi `(` e `)`, Java capisce che si tratta di una funzione. La chiamata `getServer()` restituisce qualcosa (che in effetti è un `Server`). Ottenuto questo, puoi chiamare una qualsiasi delle funzioni `broadcastMessage Server`. In questo caso, chiamiamo la funzione di `Server()`, passando un argomento di stringa chiamato `msg`. Vediamo questa riga in azione:

Per essere più chiari, puoi suddividere questa catena di chiamate in pezzi distinti, così:

```
instance = Canary.instance();
server = instance.getServer();
server.broadcastMessage(msg);
```

Visto? È un set di chiamate di funzione. Mi segui? Ecco ora un esempio che definisce una nuova funzione chiamata `castIntoBlackHole`. Osserva attentamente, perché poi farai qualcosa del genere da solo:

```
public static void castIntoBlackHole(String playerName)
{
    // Fai qualcosa di interessante con il giocatore in questo punto...
}
```

C'è più carne al fuoco qui che nell'esempio della torta. Vediamo cosa significa tutto questo.

- `public` indica che la funzione può essere usata in qualsiasi parte del programma, che è il nostro caso.
- `static` indica che puoi chiamare questa funzione di per sé e non come un plug-in (vedremo la differenza e cosa vuol dire nel prossimo capitolo).
- `void` indica che questa funzione eseguirà un paio di istruzioni ma non darà un risultato, cioè non restituirà alcun valore al chiamante.
- `castIntoBlackHole` è un nome creato da noi; è il nome della funzione, e i caratteri `()` indicano che si tratta di una funzione e che prenderà gli argomenti che abbiamo elencato. Le parentesi ci vogliono sempre, anche se la funzione non

prende nessun argomento.

In questo caso, prende un argomento che abbiamo chiamato `playerName`, che ci si aspetta sia una `String`. Per ogni argomento accettato dalla funzione, devi specificare un nome di variabile e il suo tipo. Una funzione può prendere più argomenti; per separare ogni coppia tipo-variabile si usa una virgola, come abbiamo visto in `helloCommand` nel plug-in `HelloWorld`.

Il codice di questa funzione va collocato tra parentesi graffe (`{ }`). Puoi inserire tutto il codice che vuoi, ma è buona norma non superare le 30 righe. Più corto è, meglio è; se scrivi funzioni troppo lunghe, sarai costretto a scomporle in molte funzioni più piccole per rendere il codice più leggibile.

Ecco un esempio di funzione che restituisce un valore; il suo scopo è quello di triplicare il numero che le fornisci:

```
public static double multiplyByThree(double amt)
{
    double result = amt * 3.0;

    return result;
}
```

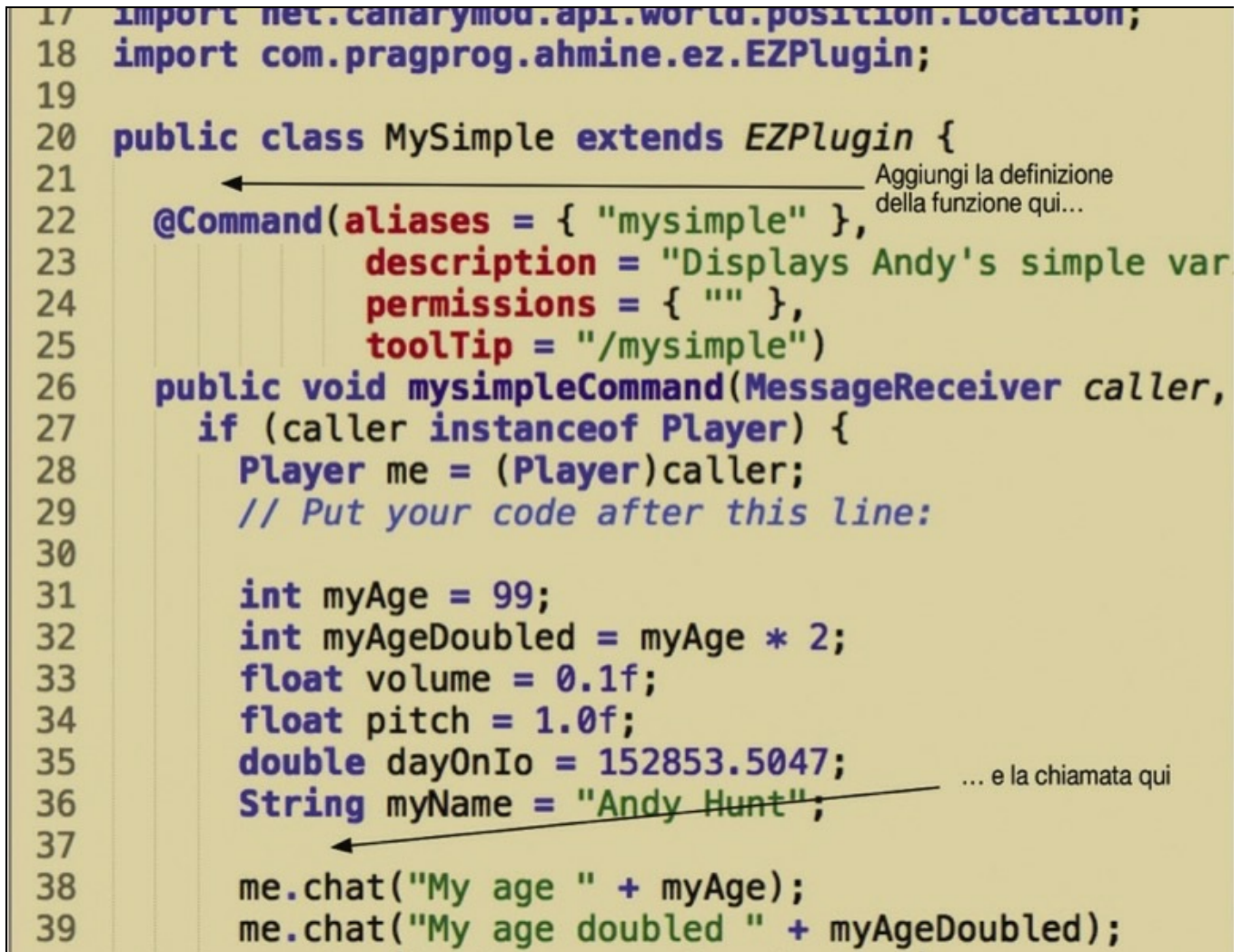
La funzione calcola un risultato e usa la parola chiave `return` per restituire il valore al chiamante. Chiama la funzione `multiplyByThree` e assegna il valore restituito alle variabili in questo modo:

```
double myResult = multiplyByThree(10.0);

double myOtherResult = multiplyByThree(1.25);
```

Ora `myResult` sarà 30.0 (`10.0` × 3), mentre `myOtherResult` sarà 3.75 (`1.25` × 3).

Prova da solo

Ora scriverai una funzione chiamata `howlong()` per calcolare quanti secondi sono passati da quando sei nato:

```
public static long howlong(int years) {
  // Scrivi questa funzione...
}
```

La funzione prende un numero di anni e restituisce un numero di secondi `long` (che è come un `int` più grande). Imbrogliamo un po' per rendere tutto più semplice, e convertiamo gli anni in secondi. (In altre parole, la funzione moltiplica il numero di anni per il numero di giorni in un anno, moltiplicato per il numero di ore in un

giorno, moltiplicato per il numero di minuti in un’ora e infine moltiplicato per il numero di secondi in un minuto.)

Aggiungi questa nuova funzione al plug-in `Simple` e chiamala per ottenere il valore, così come hai fatto per il tuo nome e la tua età.

Definisci la funzione nel punto indicato con una freccia in alto nella figura.

```
17  import net.canarymod.api.world.position.Location;
18  import com.pragprog.ahmine.ez.EZPlugin;
19
20  public class MySimple extends EZPlugin {
21
22    @Command(aliases = { "mysimple" },
23             description = "Displays Andy's simple var
24             permissions = { "" },
25             toolTip = "/mysimple")
26    public void mysimpleCommand(MessageReceiver caller,
27      if (caller instanceof Player) {
28        Player me = (Player)caller;
29        // Put your code after this line:
30
31        int myAge = 99;
32        int myAgeDoubled = myAge * 2;
33        float volume = 0.1f;
34        float pitch = 1.0f;
35        double dayOnIo = 152853.5047;
36        String myName = "Andy Hunt";
37
38        me.chat("My age " + myAge);
39        me.chat("My age doubled " + myAgeDoubled);
```


Poi aggiungi la chiamata alla funzione `howlong` dove indicato dalla seconda freccia. Assegnala a un integer grande (un `long`) e passale un’età (qui useremo `10`), così:

```
long secondsOld = howlong(10);
```

Infine, mostra tutto a video al giocatore come fa il resto delle chiamate `chat()`.

Se compilo e installo con `./build.sh`, fermo il server e lo riavvio e poi eseguo il comando `/simple` in Minecraft, il mio test con l’età di 10 anni darà come risultato 315.360.000 secondi:

```
$ cd Desktop
$ cd Simple
$ ./build.sh
Compiling with javac... Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Hai ottenuto la mia stessa risposta? Trovi il codice sorgente completo in `code/Simple2/src/simple2/Simple2.java`.

Nota che ci sono due modi diversi per lavorare con questa funzione; come spesso accade, non c'è un solo metodo "giusto" per scrivere il codice.

Siamo partiti bene, ma Java è qualcosa di più che semplici variabili e funzioni. Il linguaggio ha infatti alcune *parole chiave* speciali che puoi usare per determinare come e quando eseguire vari segmenti di codice. Ne abbiamo già viste alcune che descrivono il codice, come `public` e `static`. Ora vedremo quelle che ti permettono di controllare come il codice viene eseguito: `if`, `for` e `while`.

Usare un ciclo for per ripetere il codice

I computer sono molto più bravi degli uomini quando si tratta di compiere operazioni ripetitive; puoi dirgli di fare qualcosa dieci volte in una riga e quello lo farà esattamente così per dieci volte, oppure per cento o un milione di volte, a seconda di quello che ti serve. Un modo per fare la stessa cosa per un tot di volte è usare un ciclo `for` (*per*). `For` è una parola chiave di Java che ti permette di eseguire un ciclo in una sezione di codice per un numero fisso di volte.

In Java, il ciclo `for` è una struttura di controllo di base, ovvero un modo per controllare l'ordine di esecuzione delle righe di codice. Se devi creare un gruppo di blocchi o generare molti Creeper in uno dei mondi di Minecraft, puoi usare `for`. Lo stesso se devi passare tra tutti i giocatori attualmente online (anche se per farlo ci sono modi migliori, come vedremo in seguito).

Per esempio, il prossimo segmento di codice tratto da un plug-in genera 10 maiali nel luogo in cui ti trovi:

```
//... da qualche parte in un plug-in:
for (int i=0; i < 10; i++) {
spawnEntityLiving(location, EntityType.PIG);
}
```

In questo caso, l'istruzione `for` eseguirà le istruzioni tra parentesi graffe per 10 volte, quindi `spawn` sarà chiamata 10 volte, creando 10 maiali.

Tra le parentesi tonde in `for` ci sono tre elementi, separati da punti e virgola. Vediamo cosa fanno.

int i=0;

La prima parte dichiara e *inizializza* la variabile del ciclo. Qui è stato usato `i` come contatore del ciclo, partendo da `0` (più avanti vedremo perché). Java conta sempre partendo da 0.

i < 10;

Il ciclo viene *testato*. Ci viene detto per quanto proseguire con il ciclo e, ancora più importante, quando dobbiamo interromperlo. Il ciclo continuerà a eseguire il codice tra le parentesi graffe che seguono fino a quando `i` sarà minore di 10. Adesso come adesso questo vorrebbe dire che il ciclo continua all'infinito, quindi

ci serve l’ultima parte.

i++

L’*incremento* del ciclo. È la parte che lo fa proseguire. Qui incrementiamo ogni volta di 1 la variabile `i` nel ciclo. Ricorda: `i++` è una forma abbreviata di `i=i+1`. Si prende il valore di `i`, si aggiunge 1 e lo si risalva come nuovo valore di `i` (puoi usare questa scorciatoia ovunque, non solo nei cicli).

Usare un’istruzione if per prendere decisioni

Un’istruzione `if` (*se*) consente di prendere decisioni nel codice ed eventualmente eseguire un segmento di codice se una condizione è vera. È in questo modo che fai “pensare” il computer .

È come nel mondo reale. `If` la tua età è >= 18 anni, puoi prendere la patente. `If` la porta non è chiusa a chiave, puoi aprirla, altrimenti non puoi. `If` il comando impartito a Minecraft è uguale a `“hello”`, allora puoi inviare un messaggio. `If` la tua temperatura corporea scende a zero, sei morto.

In Java appare così:

```
if (something) {
// esegue il codice...
}
```

La parte tra parentesi (`something`) può essere qualsiasi cosa che risulta essere `true` (cioè vera) o `false` (cioè falsa); in altre parole, si tratta di una condizione booleana (ne parleremo meglio tra poco).

Se ti serve, puoi anche far eseguire un codice se qualcosa è `true` e in più specificare cosa eseguire se qualcosa è `false`:

```
if (something) {
// esegui questo codice se something è true, cioè vero
} else {
// esegui questo codice se something è false, cioè falso
}
```

Per esempio, questo è un frammento di codice che dice qualcosa di diverso a seconda che la `String` nella variabile `myName` contenga Notch o meno:

```
if (myname.equalsIgnoreCase(“Notch”))  {
    say(“Greetings from Notch!”);
} else {
    say(“Notch isn’t here anymore.”);
}
```

In programmazione, le istruzioni `if`, con o senza `else` (*altrimenti*), sono fondamentali: rappresentano il modo in cui fai sì che il computer prenda decisioni e “pensi”. Roba grossa, ma molto semplice da usare.

Fare confronti con le condizioni booleane

Le istruzioni `if` decidono cosa eseguire quando qualcosa è `true`. Ma solo quando è `true`?

Oltre che di numeri e stringhe, una variabile può anche tenere traccia di qualcosa come “attivo”. In Minecraft, utilizziamo questo espediente per capire se un giocatore è sul terreno, se una stringa coincide con un’altra, per decidere se usare un evento di gioco o ignorarlo e altro ancora.

Java chiama questo tipo di variabile `boolean` (da George Boole, il matematico inglese che elaborò questo concetto nel 1800); può essere designata come `true` o `false`, oppure puoi darle una forma matematica usando uno di questi operatori, che restituiscono tutti `true` o `false`:

`==`	Uguale a (due segni di uguale)
`!=`	Diverso da
`!`	Not (`not true` è falso, `not false` è vero)
`<`	Minore di

>	Maggiore di
<=	Minore di o uguale a
>=	Maggiore di o uguale a
&&	And (true se entrambe le cose sono vere)
||	Or (true se una delle due cose è era vera)

Per esempio, date le variabili

```
int a = 10;

int b = 5;
String h = new String("Hello");
boolean result = true;
boolean badone = false;
```

Java imposterà i seguenti confronti:

- a == 10 è true;
- b == 6 è false;
- a < 20 è true;
- b >= 5 è true;
- a > 100 è false;
- result i è true;
- !result è false (da leggere come not result, dove not restituisce l'opposto di un valore);
- result && badone è false (da leggere come and; è vero sole se entrambi sono veri);
- result || badone è true (da leggere come or; è vero sole se uno dei due è vero).

Tuttavia, la prossima riga non farà quello che ti aspetti; non sarà true:

```
h == "Hello"; // Beccato!
```

Nel caso di stringhe e oggetti, infatti (ne parleremo nel prossimo capitolo), devi utilizzare la funzione equals invece del doppio segno di uguale (==), così:

```
h.equals("Hello");    // è true
h.equalsIgnoreCase("hELLO"); // è true
```

Usare un ciclo while in base a una condizione

Usiamo un ciclo `for` quando dobbiamo eseguire del codice per un numero fisso di volte. Ma come fare se non siamo certi di questo numero? E se volessimo eseguire un ciclo finché una data condizione è vera? In questo caso utilizzeremo un ciclo `while`. Un ciclo `while` (*mentre*) continuerà a eseguire un segmento di codice fintanto che la condizione booleana è vera:

```
while (stillHungry) {
    // ...
    // Qualcosa di meglio ha reso stillHungry false!
}
```

In un certo senso è un incrocio tra un'istruzione `if` e un ciclo `for`: il ciclo continua esattamente come con `for`, ma lo fa fino a quando la condizione è vera, verificandola come fa `if`.

Quindi, se dimentichi di modificare il valore come `false`, `while` continuerà all'infinito, e il server di Minecraft si bloccherà finché non correggerai l'errore, non riavvierai il server o finché non andrà via la corrente.

Prova da solo

È il momento di creare il tuo primo ciclo. Torniamo a `MyHouse.java`, in *~/Desktop/code/BuildAHouse/src/buildahouse*, ma invece di costruire una sola casa con questa chiamata:

```
BuildAHouse.buildMyHouse(width, height);
```

scrivi un ciclo `for` che girerà per 10 volte, con la chiamata a `buildMyHouse` nel corpo. In questo modo costruirai 10 case, una mini-città!

Modifica `MyHouse.java` e aggiungi il tuo ciclo `for`, quindi costruisci e installa il plug-in come al solito:

```
$ cd Desktop
$ cd code
$ cd BuildAHouse
$ ./build.sh
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Ferma il server e riavvialo, connettiti con il tuo client e digita nuovamente

`/buildahouse`. Ora hai un nucleo di 10 case!

Per continuare

In questo capitolo hai imparato qualcos'altro sulla sintassi di Java, dai vari tipi di parentesi ai punti e virgola. Adesso sai come dichiarare le variabili e come usarle per salvare informazioni importanti. Puoi scrivere funzioni che agiranno sui tuoi dati e puoi controllare le funzioni con le istruzioni `if`, `for` e `while`.

Di seguito vedremo cosa accade quando si assemblano variabili e funzioni per creare degli oggetti, il cuore di un sistema esteso come quello di Minecraft. Gli oggetti di Minecraft ti permettono di creare plug-in per manipolare tutto quanto si trova nel suo ambiente, dai Creeper alle mucche. Continua a seguirmi.

La tua toolbox

21%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.

I plug-in contengono oggetti

Potresti aver sentito parlare di programmazione orientata agli oggetti, o sentito dire che Java è un linguaggio orientato agli oggetti. Questo sarà l'argomento del capitolo: come utilizzare gli oggetti per rappresentare gli elementi in Minecraft, dai giocatori alle mucche.

Nel codice puoi fare tutto quello che puoi fare nel gioco, e anche altro. Il nostro lavoro si concentrerà su tre tipi di oggetti: blocchi, item ed entità.

In Minecraft, ogni cosa è un oggetto

Il mondo di Minecraft è pieno zeppo di *blocchi* cubici, distribuiti ovunque. Un blocco può essere fatto da aria o da altri materiali. I blocchi sono presenti nel mondo e nell'inventario di un giocatore; tutto quello che si trova nell'inventario è un *item* (cioè un elemento).

In questo mondo fatto a blocchi hai a disposizione delle *entità*, che includono giocatori, Creeper e mucche. Anche gli item in movimento sono delle entità: una freccia in volo, una palla di neve o una pozione che viene rovesciata. E tutti sono oggetti.

Tutto quanto si trova nei nostri plug-in è un oggetto: luoghi, blocchi, entità, mucche, Creeper, giocatori e anche il plug-in stesso. Tutti oggetti, sempre.

Qui inizia il divertimento. Hai variabili che contengono dati e istruzioni espresse come funzioni, comprese alcune dichiarazioni di controllo che ripetono bit di codice o prendono decisioni. Adesso vedremo come assemblare tutto ciò in oggetti.

Prova da solo

Diamo una piccola dimostrazione di cosa sono gli oggetti in Minecraft. Nel codice che hai scaricato trovi un plug-in chiamato `NameCow`. Installalo, come mostrato di seguito:

```
$ cd Desktop
$ cd code/NameCow
$ ./build.sh
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Arresta il tuo server e poi riavvialo, connettiti con il client e potrai eseguire il nuovo comando `namecow`. Questo comando genererà una nuova mucca (*cow*) e le assegnerà un nome. Nel client di Minecraft, digita il comando seguito dal nome di quella mucca:

```
/namecow Bessie
/namecow Elise
/namecow Babe
```

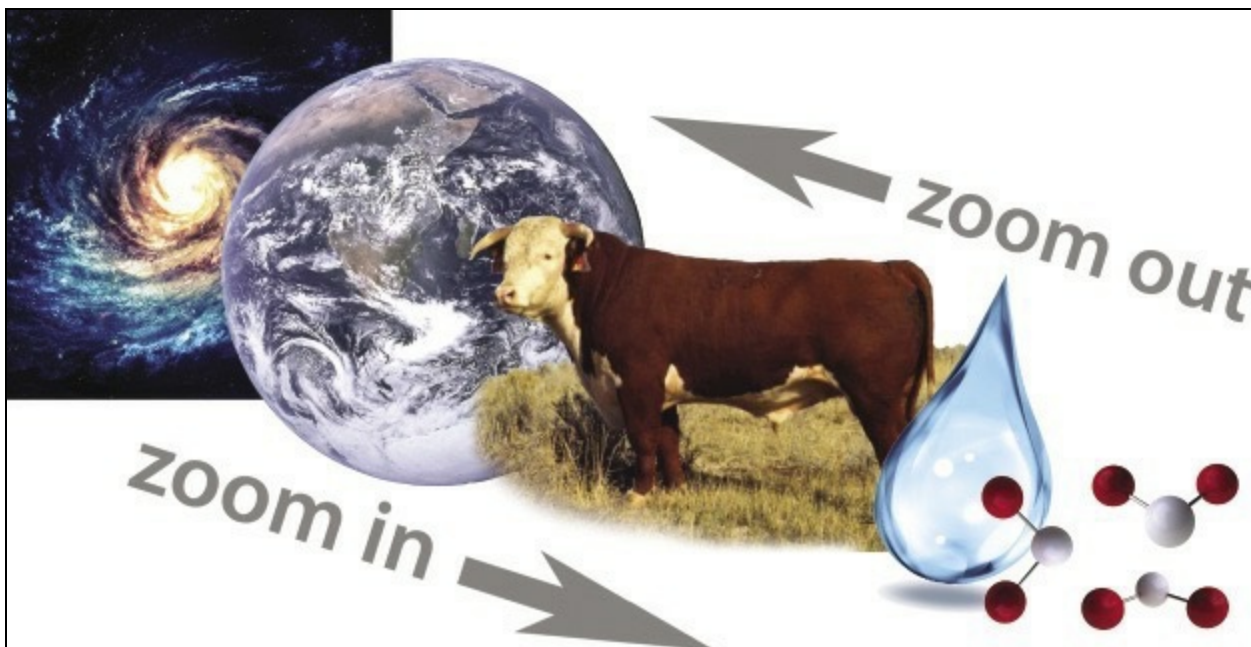
Vedrai queste mucche apparire nel gioco, ognuna identificata con il suo nome.

Questo plug-in crea un nuovo oggetto `Cow` ogni volta che lo esegui. Ogni mucca che generi ha variabili interne che tengono traccia delle sue caratteristiche specifiche: il suo nome, la sua posizione nel mondo di Minecraft, il suo stato di salute e così via. È vero che è importante che ciascuna mucca sia rappresentata da un oggetto, ma questa non è una ragione sufficiente per usare gli oggetti. C'è un motivo più profondo.

Perché preoccuparsi di utilizzare gli oggetti?

Immagina di essere il dio del tuo universo. Hai passato un'eternità a sistemare quark, atomi e molecole fino ad arrivare a pianeti e galassie, proprio come volevi tu. Ora tutto gira, e tu sei responsabile di ogni piccola particella subatomica nell'intero universo. Per quanto tu sia una divinità, gestire contemporaneamente ogni elettrone, quark o molecola è un lavoraccio (ed è anche parecchio noioso).

Come alternativa, puoi affrontare i problemi secondo la scala in cui si verificano. Se hai un inconveniente con un pianeta che si trova nel posto sbagliato, puoi spostare il pianeta; se devi giocherellare con una galassia, giocherella con quella, non con tutti i pianeti, e certamente non con tutte le forme di vita esistenti su ciascun pianeta di ogni sistema.

Per quanto possa sembrare elaborato, creare un programma è proprio questo. Sei il creatore del tuo piccolo universo privato. Magari non sei esattamente una divinità, ma devi affrontare comunque la gestione di ogni cosa a un livello molto basso – come atomi e molecole – e a un livello molto alto, come creature, montagne, pianeti e galassie. Tutto simultaneamente. Quando programmi, spesso ti troverai a focalizzarti (*zoom in*) sugli atomi per poi allargare la tua attenzione (*zoom out*) sulle galassie. Sono tutti connessi e tutto ha un senso.

Ecco perché scriviamo il codice usando gli oggetti. È un modo di organizzare i dati (in variabili) e il comportamento (in funzioni) di modo che quando dobbiamo fare qualcosa con una mucca, per esempio, possiamo trattarla come mucca "intera", e non siamo costretti a lavorare con i milioni di atomi che la costituiscono (lo stesso vale per un bioma, un mondo o una torcia).

Ancora meglio, possiamo scrivere del codice in modo che solo una `Cow` abbia i dati e le funzioni che le servono. Non c'è niente di peggio che avere le funzioni più disparate che sbucano da ogni dove; potresti finire con l'ottenere una torcia che muggisce o una mucca che fa luce… Ancora peggio, potresti ritrovarti con un mucchio di funzioni dove non sei nemmeno sicuro che potrebbero operare, oltre a migliaia di dati.

Per esempio, in Minecraft un oggetto `Cow` ha molte funzioni che puoi chiamare. Vediamone alcune:

- `teleportTo(Location` *posizione*`)`
- `setFireTicks(int` *tick*`)`
- `setAge()`
- `getAge()`
- `wingArm()`

Alcune variabili dell'oggetto tengono traccia del suo stato interno: la posizione di quella singola mucca, se va a fuoco, la sua età e così via.

Questo è tutto ciò che puoi fare con un oggetto `Cow`, ma sono cose che non hanno molto senso per un giocatore, una freccia o un albero.

Tutto quanto riguarda una mucca va nell'oggetto `Cow`, tutto quanto riguarda una freccia va nell'oggetto `Arrow` e così via. Se non procedi così, le cose diventano confuse molto in fretta, e potresti ottenere un codice che non riesci più a capire o con cui non è più possibile lavorare.

Separare, quindi, è una sana abitudine, come tenere il latte in frigorifero nel suo cartone invece di infilarlo nel sacchetto dei biscotti conservato in dispensa. (Non mescoleresti neanche le caramelle con i corn-flakes, per esempio.)

Tenere le cose distinte in oggetti diversi è la parte facile.

La parte più difficile della programmazione è dover gestire

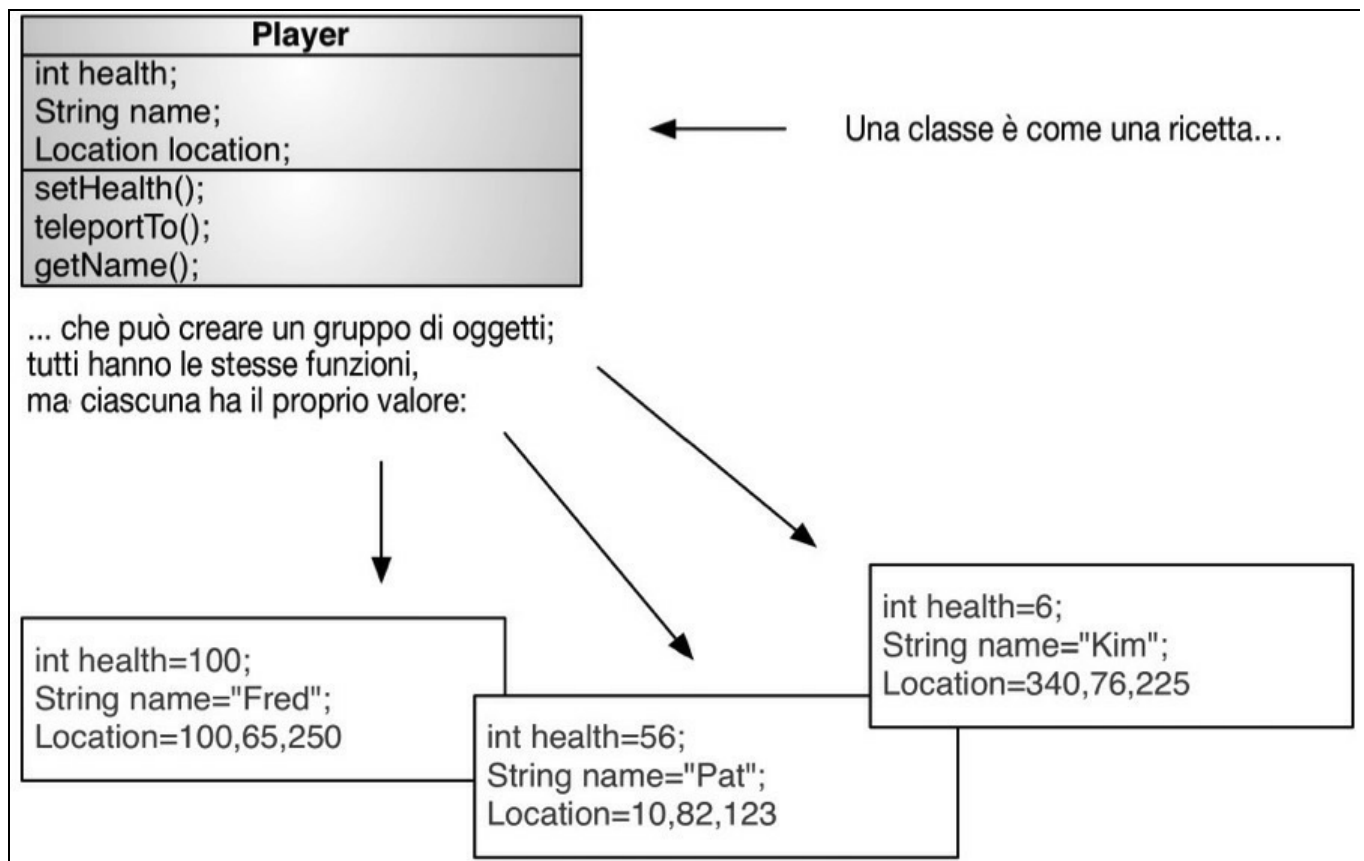
contemporaneamente ogni dettaglio al livello più basso e più alto. Parliamo di *livelli di astrazione.* Quando scrivi un gioco per gestire un giocatore in Minecraft, non gestisci la persona fisica che sta giocando. Da qualche parte c'è qualcuno in carne e ossa che è seduto davanti al computer a giocare, sudando, mangiando patatine e ascoltando la musica a tutto volume. Il tuo codice è una rappresentazione astratta del giocatore della vita reale, un'astrazione che include i dati e il comportamento necessari per il gioco. Proprio come nella vita vera, ogni astrazione contiene delle "parti". Puoi decidere di concentrati su molecole o pianeti o, nel software, su oggetti molecola e oggetti pianeta. Puoi spostarti avanti e indietro tra i vari livelli a seconda di quello che ti serve e lavorare con le parti che vuoi.

Diamo un'occhiata più da vicino a tutto questo in Minecraft e Java.

Combinare dati e istruzioni in oggetti

Immagina di dover scrivere un gioco come Minecraft da zero. Ti servirà un gruppo di giocatori; ognuno avrà il suo nome, il suo inventario, i suoi punti salute e così via, ma la struttura di ciascun giocatore sarà la stessa. Ovvero, ogni oggetto giocatore avrà lo stesso insieme di variabili (nome, stato di salute, posizione) e le stesse funzioni che potresti voler eseguire (impostare il suo livello di salute, teletrasportarlo in un nuovo luogo…).

È qui che entrano in gioco gli oggetti. Devi creare un oggetto in Java che rappresenti ciascun giocatore nel sistema. Scriverai un codice che faccia fare le cose a un giocatore, e quel codice funzionerà a prescindere dal giocatore specifico che stai usando quel momento. Questo è esattamente quello che hanno fatto i creatori di Minecraft. Ma come funziona la magia?

In Java, puoi definire un insieme di variabili e un segmento di codice che le utilizza, come in una ricetta. Dopodiché puoi creare e impiegare un oggetto costruito con quella ricetta. Java chiama questo tipo di ricetta una *classe*, ed è da lì che fa funzionare gli oggetti. Guarda la prossima figura; sono presenti alcune variabili e funzioni per la classe `Player` (giocatore) e alcuni degli oggetti che può creare.

È come costruire un oggetto con i mattoncini Lego: tutti i mattoncini possono essere uguali, ma se segui le istruzioni sulla scatola, puoi realizzare un'astronave.

Minecraft conserva alcuni dati e funzioni interessanti relativi a ogni giocatore che si trova online.

La sua *ricetta* è definita dalla classe `net.canarymod.api.entity.living.humanoid.Player`, che costruisce gli oggetti da un singolo `Player`.

Nota che prima di poter usare `Player` nel codice, devi aggiungere un'istruzione di importazione all'inizio del file, in questo caso `import net.canarymod.api.entity.living.humanoid.Player`.

Come accedere a uno di questi oggetti giocatore?

Il server di Minecraft sa chi è online in un dato momento, quindi puoi chiedere al server un elenco di oggetti `Player`; oppure puoi chiedergli un utente specifico con il suo nome, e il server ti darà un unico oggetto `Player`. Ecco come.

In un plug-in, puoi accedere al server usando la funzione chiamata `Canary.getServer()`, che ti fornisce un oggetto che rappresenta il server. Una volta ottenuto l'oggetto server, puoi richiedergli il giocatore che ti interessa. Usa

getPlayer e chiamalo con il nome del giocatore.

La sequenza di istruzioni è simile a quella riportata di seguito (la analizzeremo prima pezzettino per pezzettino, poi termineremo con l'esempio completo). Si inizia con le istruzioni import che definiscono le ricette di classe che userai:

```
//Dall'alto
import net.canarymod.Canary;
import net.canarymod.api.Server;
import net.canarymod.api.entity.living.humanoid.Player;
```

Successivamente, nel tuo codice del plug-in, puoi usare le variabili di questo tipo (Player e Server):

```
Server myserver = Canary.getServer();
Player fred = myserver.getPlayer("fred1024");
```

Presumendo che fred1024 sia online in quel momento, abbiamo una variabile chiamata fred che rappresenta l'oggetto Player. Se Fred non fosse online, la variabile fred sarebbe uguale al valore speciale null, il modo in cui Java indica che "non c'è nessuno"; in altre parole, fred è impostato a nessun oggetto. (Se fred è null e provi a eseguire una funzione da fred, come fred.isSneaking(), ottieni un errore e il plug-in si blocca.)

Gli oggetti sono costituiti da parti. Contengono delle cose. Un int, come 5, è semplicemente il numero cinque. Non può fare altro; non può memorizzare niente. È quello e basta. Tuttavia, gli oggetti hanno funzioni e variabili a cui puoi accedere con un punto (.).

ISTRUZIONI DI IMPORTAZIONE

Ogni ricetta di oggetto in Java è contenuta in un package, come java.util, net.canarymod.Canary o qualcosa del genere. Devi dichiarare questo package in un'istruzione di importazione all'inizio del file del codice sorgente Java.

Se ti dimentichi di farlo, ottieni un errore da javac che dice "cannot find symbol" ("non riesco a trovare il simbolo").

Consulta la documentazione di Canary o di Java per trovare il nome completo del package. Nei nostri esempi useremo alcuni dei più comuni, per Player, Server, Location, Entity e diverse librerie Java. Per tua comodità, trovi un elenco nell'Appendice F.

Con fred a portata di mano, puoi ottenere e impostare i dati per quel giocatore usando il punto (.) per indicare quale funzione o variabile vuoi avere nell'oggetto. Ecco alcuni frammenti di codice che rendono l'idea:

```
// Fred si sta avvicinando?
boolean sneaky = fred.isSneaking();
```

```
// Ha ancora fame?
int fredsHunger = fred.getHunger();

// Tienilo a stecchetto!
fred.setHunger(0);

// Dov’è Fred?
Location where = fred.getLocation();
```

L’oggetto `Player` ha una funzione chiamata `isSneaking()`, che restituisce `true` o `false` a seconda che il giocatore si stia o meno avvicinando.

Ricorda, questo è il tipo di cose che puoi usare in un’istruzione `if`:

```
if (fred.isSneaking()) {
    fred.setFireTicks(600); // Dagli fuoco!
}
```

C’è anche una funzione `getHunger()` che restituisce un `int` che ti dice quanto il giocatore è affamato, e puoi anche impostare il suo livello di fame. L’ultimo segmento di codice mostra come ottenere la posizione corrente del giocatore nel mondo.

Come puoi intuire, gli oggetti contengono variabili interne con cui lavorano le loro funzioni. In questo esempio, l’oggetto `Player` per Fred memorizza un luogo al suo interno. Possiamo ottenere il valore della posizione di Fred o impostare un nuovo valore, ma Fred continuerà a mantenere una copia interna della sua posizione corrente.

Puoi anche compiere alcune operazioni interessanti, per esempio eseguire dei comandi come se tu fossi Fred:

```
fred.executeCommand(“tell mary179 I love you”)
```

Grazie a questa riga, Mary penserà che Fred le abbia mandato un messaggio d’amore. Giochiamo con la `Location` (la posizione) di Fred e vediamo quali altri dispetti possiamo inventarci:

```
Location where = fred.getLocation();
```

Ora la variabile che abbiamo chiamato `where` punterà a un oggetto `Location` che rappresenta la posizione in cui si trova Fred nel mondo.

Creare gli oggetti

Se vuoi, puoi creare una posizione nuova da zero:

```
double x, y, z;
x = 10;
y = 0;
```

```
z = 10;
Location whereNow = new Location(x, y, z);
```

e fare la stessa cosa in un solo passo:

```
Location whereNow = new Location(10, 0, 10);
```

In Java, un nuovo oggetto si crea usando la parola chiave `new`. Quando ricorri a `new` per realizzare un oggetto, Java creerà quell'oggetto per te ed eseguirà il suo *costruttore*, una funzione che ha lo stesso nome della classe (come `public Location()`). Il costruttore ti dà l'opportunità di impostare tutto quello che serve nell'oggetto. Non restituisce nulla e non è dichiarato con un tipo di ritorno; Java restituisce automaticamente il nuovo oggetto dopo che l'hai impostato.

Ci occuperemo di come creare le definizioni degli oggetti più avanti nel libro; per ora, utilizzeremo quello che ci ha fornito Minecraft insieme allo scheletro del nostro plug-in.

Decisa una posizione, puoi mandarci Fred:

```
fred.teleportTo(whereNow);
```

Di colpo Fred si troverà… a soffocare nella roccia del sostrato (perché in questa posizione `y` è zero). Ahi.

Plug-in: PlayerStuff

Facciamo ancora qualche esperimento con `Player`.

Adesso installeremo il plug-in `PlayerStuff` e modificheremo alcune delle proprietà di oggetto per i giocatori nel mondo di Minecraft.

Nell'oggetto `Player` ci sono tutti i tipi di funzioni interessanti che ti permettono di ottenere informazioni sul giocatore e di impostarne i valori. Vediamole.

- `chat()`
- `getWorld()`
- `getDisplayName()` (si può impostare)
- `getExperience()` (si può impostare)
- `getHunger()` (si può impostare)
- `getHealth()` (si può impostare)
- `isSleeping()`
- `getLocation()`

Possiamo inviare un messaggio a un giocatore, ottenere dei valori e altro ancora. Di seguito è riportato il codice per un plug-in completo che dimostra alcune di queste caratteristiche; come vedrai, fornisce il comando `"/whoami"`. Nel caso di plug-in più avanzati, questo approccio potrebbe essere l'ideale per effettuare il debug degli oggetti del gioco visualizzando le informazioni su di essi. Quando si scrive del codice e questo non funziona, puoi stampare un paio di valori di variabili diverse per capire cosa sta succedendo.

Ecco il codice, che si trova nella cartella del plug-in *Desktop/code/PlayerStuff*. Il plug-in completo contiene il file `Canary.inf` che ti serve.

PlayerStuff/src/playerstuff/PlayerStuff.java

```
package playerstuff;
import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.api.world.position.Location;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import com.pragprog.ahmine.ez.EZPlugin;

public class PlayerStuff extends EZPlugin {
```

```
  @Command(aliases = { "whoami" },
            description = "Displays information about the player.",
            permissions = { "" },
            toolTip = "/whoami")
  public void playerStuffCommand(MessageReceiver  caller, String[] parameters) {
      if (caller instanceof Player) {
        Player me = (Player)caller;
        String msg = "Your display name is " + me.getDisplayName();
        me.chat(msg);
        me.getWorld().setRaining(true);
        me.getWorld().setRainTime(100); // 5 secondi
        float exp = me.getExperience();
        int food = me.getHunger();
        float health = me.getHealth();
        Location loc = me.getLocation();

        me.chat("Your experience points are " + exp);
        me.chat("food is " + food);
        me.chat("health is " + health);
        me.chat("you are at " + printLoc(loc));
        me.chat("water falls from the sky ");
    }
  }
}
```

Installalo adesso:

```
$ cd Desktop
$ cd code/PlayerStuff
$ ./build.sh
```

Riavvia il server e riconnetti il tuo client di Minecraft.

Cosa accade quando digiti il comando `"/whoami"` dal client di Minecraft? Ecco quello che vedo io:

Studiamo il codice passo. Per iniziare, otteniamo l'oggetto giocatore `me`. Usando `me`, otteniamo poi il nome del giocatore, al quale inviamo un messaggio.

Facciamo piovere (o nevicare) sul giocatore impostando la pioggia (*rain*) a `true` e facendola durare 5 secondi usando `setRainTime(100)` (per far smettere di piovere, imposta la variabile a `false`). Ci sono 20 tick del server ogni secondo, quindi 100 tick corrisponderanno a circa 5 secondi. Adesso otterremo i punti esperienza per il livello successivo e il livello cibo, e li invieremo sotto forma di messaggi al giocatore. Puoi giocare in questo modo per un po' ed eseguire "`/whoami`" per vedere se il tuo cibo e la tua esperienza cambiano in qualche modo.

È semplice: `me` è un oggetto di tipo `Player`, e possiamo ottenere valori dai vari giocatori e inviare comandi a `me` per compiere azioni da giocatore.

Ecco a cosa servono gli oggetti.

Prova da solo

Nella figura precedente avrai notato che appare anche una riga che dice se stai o no dormendo (*sleeping*). Aggiungi una variabile locale `boolean` in `PlayerStuff` e impostala a `true` o a `false` a seconda che il giocatore stia dormendo o meno, quindi crea un messaggio opportuno per mostrarlo a video.

Costruisci il plug-in con `build.sh` e provalo. Puoi vedere quello che ho fatto io in `code/MyPlayerStuff/src/myplayerstuff/MyPlayerStuff.java`.

Per continuare

In questo capitolo hai visto come usare gli oggetti Java: come importare un package Java e una classe, come usare `new` per creare gli oggetti e come modificare le proprietà degli oggetti che influiranno sul gioco. Tutto questo ci servirà per i prossimi capitoli.

Nelle prossime pagine daremo un'occhiata più da vicino a come i plug-in vengono inseriti in Minecraft, come aggiungere i comandi e come trovare le cose nel mondo di Minecraft.

La tua toolbox

28%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.
- Usare gli oggetti Java.
- Usare le istruzioni `import` per i package Java.
- Usare `new` per creare gli oggetti.

Aggiungere un comando di chat, posizioni e bersagli

Ora sai che un giocatore è rappresentato dall'oggetto Player e che il server è un oggetto Server. Non dovrebbe quindi essere una sorpresa scoprire che gli stessi plug-in sono degli oggetti Plugin.

Canary ha definito per questo scopo una classe `Plugin` di base (una "ricetta") che lo sa. Per facilitarti le cose, abbiamo aggiunto in più una classe `EZPlugin`. Il lavoro di chi scrive plug-in è quello di fornirgli un codice che rientri in questa struttura.

Come abbiamo visto, la prima riga di un plug-in ne dichiara il nome e poi aggiunge la frase magica `extends EZPlugin`:

```
import net.canarymod.plugin.Plugin;
import com.pragprog.ahmine.ez.EZPlugin;

public class MyFavoritePlugin extends EZPlugin {
```

Questo rende `Plugin` e `EZPlugin` genitori della classe `MyFavoritePlugin`, proprio come negli esempi del Capitolo 5.

Il server di Minecraft sa già come lavorare con una classe `Plugin`, e visto che questa classe è genitore del tuo plug-in, sa anche come lavorare con il tuo plug-in, anche se questo non esisteva ancora quando Canary è stato creato. Gli basta avere da te un paio di funzioni che sa come chiamare.

Oltre che del codice, Canary ha bisogno di un file di configurazione chiamato `Canary.inf`. Hai visto una descrizione di questo file nel Capitolo 3, quando abbiamo costruito i primi plug-in. Il file fornisce al server alcune informazioni di base sul plug-in, così che possa caricarlo.

Con il file di configurazione e il codice, il server di Minecraft può eseguire il plug-in così come qualsiasi altra parte del gioco.

Plug-in: SkyCmd

Costruiremo adesso un plug-in nuovo di zecca chiamato `SkyCmd`. Al suo interno creeremo un comando `sky` che teletrasporterà tutte le creature (ma non i giocatori) di 50 blocchi verso l'alto. È molto comodo di notte, quando ci sono in giro Creeper e scheletri…

Ecco il file sorgente completo per il plug-in:

SkyCmd/src/skycmd/SkyCmd.java

```
package skycmd;
import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;

import net.canarymod.api.world.position.Location;
import net.canarymod.api.entity.living.EntityLiving;
import java.util.List;
import com.pragprog.ahmine.ez.EZPlugin;

   public class SkyCmd extends EZPlugin {

 1 @Command(aliases = { "sky" },
             description = "Fling all creatures into the air",
             permissions = { "" },
             toolTip = "/sky")
   public void skyCommand(MessageReceiver caller, String[] parameters) {
2   if (caller instanceof Player) {
3     Player me = (Player)caller;
      List<EntityLiving> list = me.getWorld().getEntityLivingList();
      for (EntityLiving target : list) {
        if (!(target instanceof Player)) {
          Location loc = target.getLocation();
          double y = loc.getY();
          loc.setY(y+50);
          target.teleportTo(loc);
        }
      }
    }
  }
 }
```

Confrontalo con il nostro file originale `HelloWorld.java`, molto più semplice. Nota che adesso l'istruzione `package` in alto e più sotto l'istruzione `public class` fanno riferimento entrambe a `SkyCmd` invece che a `HelloWorld`.

Vediamo più da vicino come un plug-in gestisce un comando della chat come `/sky`.

Gestire i comandi della chat

Il frammento di codice `@Command` al punto **1** dice che questa funzione, `skyCommand`, è responsabile della gestione del comando `/sky`. Questo vuol dire che quando il giocatore digita il comando `/sky`, verrà chiamata la tua funzione `skyCommand`.

La prima cosa da controllare è un po' complessa; quello che accade è che il `MessageReceiver` che ci viene passato qui potrebbe non essere un `Player`. Potrebbe essere un oggetto `Player` o chissà cos'altro. Dobbiamo accertarci che sia davvero un `Player`, quindi verifichiamolo direttamente al punto **2** usando la parola chiave di Java `instanceof`. Se il test è positivo e abbiamo a che fare davvero con un `Player`, possiamo eseguire il grosso del comando partendo dal punto **3**. (Se invece non è un `Player`, è probabilmente un comando da console, se li ammetti.)

La funzione `skyCommand` inizia con un altro tocco magico. Con la certezza che la variabile `caller` è davvero del tipo `Player` (quindi non semplicemente un `MessageReceiver` o qualche altro genitore o figlio), possiamo convertirla nel tipo `Player` usando un operatore di *cast*.

In questo modo l'espressione `(Player) caller` restituisce la variabile `caller` convertita con un operatore di cast al tipo `Player`, così che tu possa assegnarla alla variabile `me`. Sembra un caos, e un po' lo è, ma è anche qualcosa che puoi copiare e incollare, visto che useremo questa piccola ricetta in quasi tutti i plug-in per ottenere un oggetto `Player`.

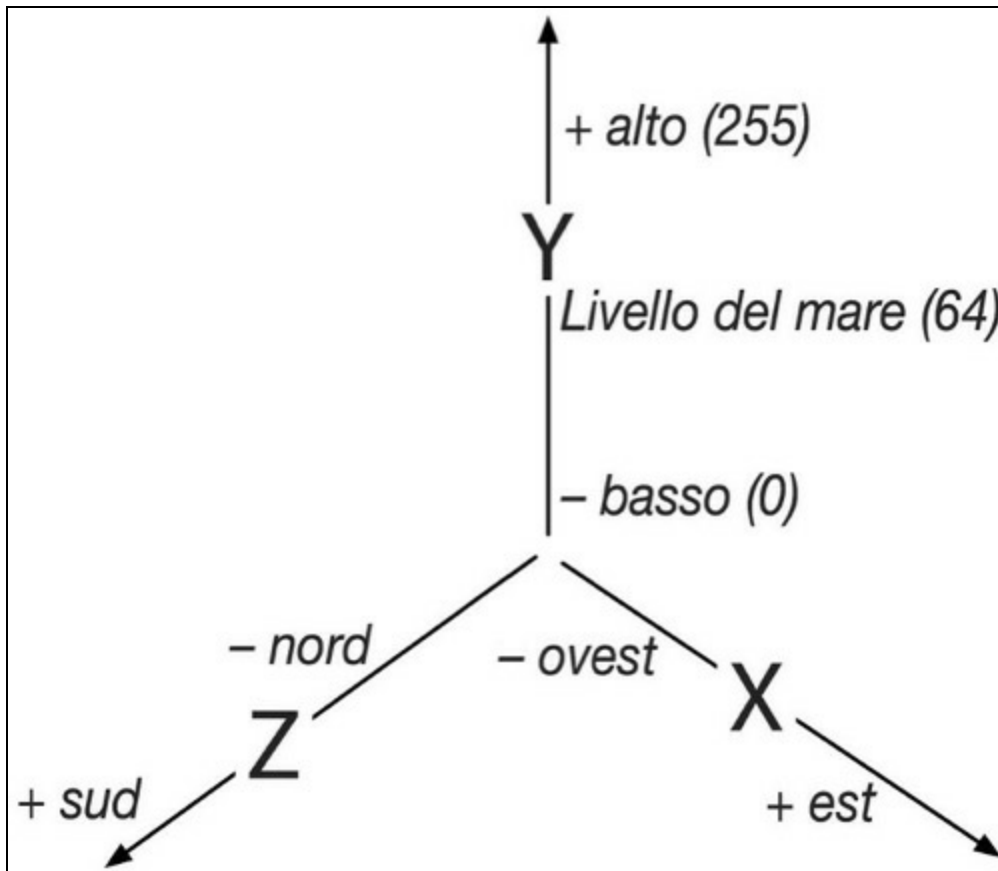
Ora che abbiamo un vero oggetto `Player` referenziato da `me`, con `me.getWorld().getEntityLivingList()` possiamo procurarci un elenco di tutte le entità viventi in questo mondo, raccolte in una `List` che percorreremo con un ciclo `for`.

Studieremo i dettagli di tutto questo nel prossimo capitolo, ma prima vediamo come funzionano gli oggetti `Location` (posizione, luogo). In questo caso, imposteremo la variabile `target` (bersaglio, destinazione) per ogni voce nell'elenco delle entità man mano che lo scorriamo con il ciclo `for`. Se il `target` non è un giocatore amico, lo lanceremo in aria, cambiando la sua posizione con `teleportTo()`.

Gli oggetti `Location` sono importanti, perché permettono di ottenere e impostare le coordinate di tutto quanto si trova in Minecraft. Vediamo come manipolarli per far

fare un bel volo alle creature.

Usare le coordinate di Minecraft

Un oggetto `Location` memorizza tre coordinate: x, y e z, come vedi nella figura.

Il valore x va da ovest (negativo) a est (positivo), la coordinata z va da nord (negativo) a sud (positivo), mentre y va dal basso (negativo) all'alto (positivo), dove un valore y pari a 0 è il livello più basso della roccia del sostrato e 64 è il livello del mare. Questo significa che per far volare verso l'alto un giocatore o un'entità, devi aggiungere qualcosa al valore y.

Otterremo il valore y corrente di ciascun target da `loc` e lo salveremo come `y`. In seguito cambieremo il valore di `loc` aggiungendo 50. Ed ecco la parte divertente: chiamando `target.teleportTo(loc)` diremo al target di teletrasportarsi in questa nuova posizione.

Wow! Così tanta roba in così poche righe di codice! Compilalo e installalo usando `build.sh` come abbiamo fatto finora:

```
$ cd Desktop
$ cd code/SkyCmd
$ ./build.sh
```

Arresta il server e riavvialo, quindi prova il nuovo comando `/sky` per divertirti un po’. Assicurati di essere in modalità Sopravvivenza e non in quella Creativa (nel gioco puoi digitare `/gamemode c` per la modalità Creativa e `/gamemode s` per quella Sopravvivenza), poi aspetta che scenda la notte e che arrivino i Creeper…

Prova da solo

Beh, è stato uno spasso! Proviamo adesso qualcosa di leggermente diverso: aggiungerai al plug-in `SkyCmd` un nuovo comando che crea dieci calamari-bomba, e lo farai tutto da solo.

Aggiungerai delle annotazioni per un comando chiamato `“squidbomb”`, e utilizzerai un ciclo `for` e una delle nostre funzioni *helper* (di aiuto).

`spawnEntityLiving` prende una posizione e quello che vuoi generare:

```
spawnEntityLiving(newloc, EntityType.SQUID);
```

`spawnEntityLiving` restituisce un’`Entity`, ma per ora non la useremo.

Per esempio, per creare un calamaro, ti serve `import net.canarymod.api.entity.EntityType` in cima al file. Successivamente, nella tua funzione, passa `EntityType.SQUID` alla funzione `spawnEntityLiving`. Così puoi realizzare una “bomba” di calamari:

```
import net.canarymod.api.entity.EntityType;

//... altre parti non mostrate

   // Crea un calamaro. Bella roba.
    for (int i = 0; i < 10; i++) {
    spawnEntityLiving(location, EntityType.SQUID);
    }
```

Usalo nel tuo nuovo comando nel plug-in `SkyCmd`. Non dimenticare di fare quanto segue.

1. Aggiungi il nuovo comando (`squidbomb`) a `SkyCmd` usando l’annotazione `@Command` come abbiamo visto in precedenza e la tua nuova funzione.
2. Ricompila e installa usando `build.sh`.
3. Arresta il server e riavvialo per applicare la modifica.

Dovrai aggiungere una nuova annotazione di comando, cha ha questo aspetto:

```
@Command(aliases = { “squidbomb” },
          description = “Drop a fixed number of squid on your head.”,
```

```
permissions = { "" },
toolTip = "/squidbomb")
```

C'è un piccolo inconveniente: tutti i calamari si dispongono uno sopra l'altro, mentre sarebbe meglio che fossero distribuiti ciascuno in una posizione diversa. In Java si può utilizzare la funzione `Math.random()`, che fornisce un numero casuale che va da 0 fino a (ma escluso) 1. Per ottenere un numero casuale tra 0 e 5, è sufficiente moltiplicare `Math.random()` per 5. Per esempio, per ottenere una nuova coordinata x puoi usare un'espressione come `loc.getX() + (Math.random() * 5)`. Quando si fanno somme e moltiplicazioni, è una buona idea usare le parentesi; in questo caso vogliamo moltiplicare un numero casuale nell'intervallo da 0 a 1 per 5, e poi aggiungerlo alla coordinata x originale.

Tocca a te adesso: migliora la "bomba" creando una nuova posizione in base alla posizione del giocatore di cui già disponi e aggiungi un po' di casualità alle coordinate z e z. Perché il calamaro cada dall'alto, aggiungi 10 alla coordinata y.

Prova a fare questo esercizio da solo. Se ti blocchi e hai bisogno di aiuto, puoi guardare il mio codice e il file di configurazione in *code/SquidBomb*, dove ho creato il plug-in completo.

Cercare blocchi o entità nei dintorni

Canary fornisce una funzione molto comoda in `net.canarymod.BlockIterator`. Un `BlockIterator` permette di trovare tutti i blocchi lungo una linea di vista nel gioco. Ancora più utile è la versione in cui passi un `LineTracer` (creato da un `Player`) e un flag booleano, che è dichiarato nell'API Canary come segue (si tratta di un'aggiunta recente a CanaryMod, e non la si trova ancora nella documentazione):

```
public BlockIterator(LineTracer tracer, boolean include_air)
```

Ottieni così un oggetto `BlockIterator` che puoi usare per individuare i blocchi lungo la linea di vista da quell'entità. Il flag `boolean` dice di includere blocchi di `Air`, e funziona così:

```
BlockIterator sightItr = new BlockIterator(new LineTracer(me), true);
while (sightItr.hasNext()) {
    Block b = sightItr.next();
    // fa qualcosa con questo blocco, b
}
```

Puoi controllare ogni blocco lungo la linea di vista di questo giocatore e trovare il primo blocco che non è `Air`: quello sarà il "target" del giocatore. Oppure puoi dar fuoco a ciascun blocco lungo il percorso e trasformare quel target in `Lava`.

Ecco un plug-in che fa proprio questo.

Plug-in: LavaVision

Questo plug-in esegue un `BlockIterator` per il giocatore e imposta un effetto fiamma. Il primo blocco non `Air` è il target, e lo imposteremo come `Lava`.

Per dare un po' di pepe al tutto, ho aggiunto un effetto a ciascun blocco che attraverseremo:

```
spawnParticle(b.getLocation(), Particle.Type.LAVASPARK);
```

`LAVASPARK` aggiunge alcune scintille lungo la linea di vista.

Ho inserito anche un effetto audio tramite la funzione helper `playSound`, che prende una `Location` e un `Sound`, cioè un suono (di cui se vuoi puoi variare il volume e il tono tramite `floats`):

```
playSound(b.getLocation(), SoundEffect.Type.EXPLODE);
```

La documentazione di Canary elenca una serie di possibili effetti per `net.canarymod.api.world.effects.SoundEffect.Type`. Io ho scelto `EXPLODE`, per fare un po' di scena. (Volendo, dopo di questo, puoi aggiungere il volume e il tono come due numeri; sono specificati come `float` e non come `double`, quindi ricorda di aggiungere il modificatore `f`.)

Ecco il codice completo, con l'iteratore e l'effetto sonoro:

LavaVision/src/lavavision/LavaVision.java

```
package lavavision;

import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.effects.Particle;
import net.canarymod.api.world.effects.Particle.Type;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.effects.SoundEffect;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.BlockIterator;
import net.canarymod.LineTracer;
import com.pragprog.ahmine.ez.EZPlugin;

public class LavaVision extends EZPlugin {

  @Command(aliases = { "lavavision" },
           description = "Explode your target into a ball of flaming lava",
           permissions = { "" },
           toolTip = "/lavavision")
  public void lavavisionCommand(MessageReceiver  caller, String[] args) {
    if (caller instanceof Player) {
```

```
      Player me = (Player)caller;

      BlockIterator sightItr = new BlockIterator(new LineTracer(me), true);
      while (sightItr.hasNext()) {
        Block b = sightItr.next();
        spawnParticle(b.getLocation(), Particle.Type.LAVASPARK);
        if (b.getType() != BlockType.Air) {
          b.getWorld().setBlockAt(b.getLocation(),  BlockType.Lava);
          playSound(b.getLocation(), SoundEffect.Type.EXPLODE);
          break;
        }
      }
    }
  }
}
```

Installa questo plug-in come fai di solito:

```
$ cd Desktop
$ cd code/LavaVision
$ ./build.sh
```

Durante il ciclo, genereremo un effetto `LAVASPARK` per ciascun blocco lungo la linea di vista. Se colpiamo qualcosa che non è `Air`, imposteremo il tipo di quel blocco a `Lava`, riprodurremo un effetto sonoro e con `break` usciremo dal ciclo.

Arresta il server e riavvialo, poi guardati in giro in Minecraft per individuare un target. Digita il comando `/lavavision` e guarda la bolla di lava.

Adesso è il tuo turno!

Modifica gli effetti utilizzati da questo plug-in. Cambia il tipo di effetto

particella da LAVASPARK a qualcos'altro di interessante. Tutti gli effetti disponibili sono elencati nella documentazione (http://bit.ly/1AvNxsk).

Successivamente, cambia l'effetto audio da EXPLODE (un'esplosione) a qualcosa di più leggero, per esempio un BURP (un… rutto). Anche gli effetti sonori sono elencati nella documentazione (http://bit.ly/1F9KULy).

Per continuare

In questo capitolo hai visto come aggiungere un nuovo comando a un plug-in. Ora puoi iniziare ad applicare le tue nuove idee ai plug-in esistenti, oppure puoi lavorare con le posizioni e i blocchi nel gioco. Tuttavia, non appena inizi a gestire più luoghi e blocchi, sorge un problema: come fa Java a memorizzare gli elenchi di questi oggetti, e come lavorare con le “pile” di dati che potresti aver bisogno di trovare per nome o secondo un dato ordine?

Nel prossimo capitolo scopriremo un comando che ci permette di ricordare le informazioni, tutto quello di cui serve tenere traccia. Parleremo ancora delle variabili in Java (chi può vederle e chi no), e soprattutto vedremo come mantenere e lavorare con tutte queste pile di dati.

In breve, vedremo come avere sempre il controllo dei dati.

La tua toolbox

36%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.
- Usare gli oggetti Java.
- Usare le istruzioni `import` per i package Java.
- Usare `new` per creare gli oggetti.
- Aggiungere un nuovo comando a un plug-in.
- Lavorare con gli oggetti `Location`.
- Trovare blocchi ed entità.

Usare pile di variabili: gli array

Ora che sappiamo come creare un comando che fa qualcosa in un plug-in, dobbiamo capire come ricordarci di tutto, cioè come usare al meglio le variabili Java per tenere traccia dei valori. Vedremo quando usare un determinato tipo di variabile e come lavorare in vari modi con le pile di dati (cose come le liste di oggetti Player); già che ci siamo, costruiremo anche un paio di plug-in.

Le variabili e gli oggetti vivono nei blocchi

Le variabili vivono all'interno di *blocchi*. Abbiamo già visto cosa sono i blocchi e come utilizzarli: sono semplici frammenti di codice scritti tra le parentesi graffe `{` e `}`.

Java è un linguaggio di programmazione "strutturato a blocchi". Discende da una famiglia di linguaggi che risale all'antico Algol degli anni Sessanta, seguendo una linea genealogica che si estende dal padre di tutti i linguaggi di programmazione, C, e dai suoi figli C++ e Objective-C, giù fino al suo figliastro e mezzo cugino Java. Dopo l'arrivo di Java, Microsoft ha pubblicato C#, che è pressoché identico a Java (ma solo per pura coincidenza).

Funzionano tutti più o meno nello stesso modo. In questi linguaggi, infatti, si lavora a blocchi di codice. Lo fanno le istruzioni `if`, per esempio. Gli stessi oggetti che definiamo (come i plug-in) sono blocchi di codice, e l'intera struttura del linguaggio si basa su blocchi di codice racchiusi tra parentesi graffe.

Qui è dove vivono le variabili. Per esempio, nel plug-in `HelloWorld`, osserva la sezione in cui creiamo e inviamo un messaggio:

```
public void helloCommand(MessageReceiver caller, String[] parameters) {
  String msg = "That'sss a very niccce EVERYTHING you have there...";
  Canary.instance().getServer().broadcastMessage(msg);
}
```

Nel corpo di questa funzione dichiariamo e assegniamo una variabile chiamata `msg`. Si tratta di una variabile *locale*, perché vive solo finché questo particolare blocco di codice è in esecuzione, racchiusa tra la sua dichiarazione e il simbolo `}`. Non puoi usarla prima di questo punto né altrove nel programma. Potrebbe esserci un'altra variabile chiamata `msg` da qualche altra parte, ma sarebbe una variabile completamente diversa, con valori differenti. Questa variabile `msg` è visibile e utilizzabile solo localmente, qui, in questo blocco di codice. Ecco perché si dice che il suo *scope*, cioè il suo ambito, è locale.

Anche i parametri che dichiari in una funzione sono considerati locali. Nella dichiarazione per `helloCommand`

```
public void helloCommand(MessageReceiver caller, String[] parameters) {
```

le variabili `caller` e `parameters` sono disponibili all'interno di questa funzione, ma

non sono visibili da nessun’altra parte. Sono locali.

La maggior parte delle volte ti serviranno solo variabili locali, che sono anche abbastanza sicure. Nessun’altra sezione del programma può usarle o modificarle, ed è ben chiaro quale riga di codice imposta il valore di una variabile locale e dove questa viene impiegata.

Variabili globali

Tuttavia, puoi creare delle variabili che hanno un ambito più esteso e che non sono esclusivamente locali (è quello che faremo per i plug-in di questo capitolo).

Per esempio, potresti aver dichiarato una variabile che può essere utilizzata da più funzioni; magari l’intera classe, o addirittura l’intero programma e tutte le librerie, possono vederla e modificarla. In questo caso abbiamo una variabile *globale*.

Ne abbiamo già utilizzate in precedenza, ma allora non te l’ho fatto notare. Sbirciamo in `EZPlugin`, che abbiamo usato nel nostro primissimo plug-in, `HelloWorld`. `EZPlugin` ha una variabile statica a livello di classe chiamata `logger` dichiarata all’inizio. (Un *logger* è una specie di “registratore” che tiene un diario dei vari eventi.)

EZPlugin/src/com/pragprog/ahmine/ez/EZPlugin.java

```
public class EZPlugin extends Plugin implements CommandListener {
  /*Metodi boilerplate per tutti i nostri plug-in */

  public static Logman logger;

  public EZPlugin() {
    logger = getLogman();
  }

  @Override
  public boolean enable() {
    logger.info (“Starting up”);
```

Chiamiamo `getLogman` e richiediamo l’oggetto `logger`. Quando ci viene restituito, lo assegniamo alla nostra variabile `logger`. Questa è di tipo `Logman`, come dovrebbe essere secondo la documentazione, ma avrai notato che a questa dichiarazione abbiamo aggiunto la parola `static` (cioè statica).

“Statica” qui significa due cose.

- Accedi alla variabile usando il nome del plug-in senza un oggetto del plug-in.

In questo caso, `HelloWorld.logger` funzionerà da qualsiasi punto, all'interno o all'esterno di un oggetto.

- La variabile statica (`logger`) è comune a tutti gli oggetti di `HelloWorld` e vive sempre, a prescindere dalla vita di qualsiasi variabile locale in qualsiasi oggetto.

Questo significa che qualsiasi funzione all'interno di un oggetto di `HelloWorld` può andare e venire, e lo stesso avviene per le rispettive variabili locali, mentre le variabili statiche rimarranno lì e ricorderanno sempre i propri valori.

Poiché `logger` è la prima cosa che dichiariamo nel plug-in, e non nella funzione di per sé, tutte le funzioni nel plug-in – cioè tutto quanto è compreso tra i caratteri `{` e `}` – potranno utilizzarla. Non è locale per nessuna delle funzioni.

Di conseguenza, puoi usare `logger` ovunque in `HelloWorld.java`:

```
...
public void myFavoriteFunction() {
    logger.info("Made it this far");
}
...
```

Osserva che non occorre nessun altro tipo di dichiarazione; puoi semplicemente usare `logger` ovunque nel plug-in che ti interessa. È un metodo molto comodo per tenere traccia di quello che sta succedendo: aggiungi un'istruzione `logger`, e potrai ottenere a video un elenco delle variabili in quel punto del codice.

Le variabili globali come queste, però, possono nascondere dei pericoli.

Perché? Proprio perché chiunque e dappertutto può modificare il loro valore anche a tua insaputa. Potresti saperlo o non saperlo. Se qualcosa va storto, sarai costretto a esaminare ogni singolo frammento di codice nel sistema per cercare di capire dove è stata impostata la variabile in modo errato e perché. È un grosso lavoro, e potenzialmente può creare confusione.

Tuttavia, potresti davvero aver bisogno di una variabile globale come questa, che non vuoi venga modificata da nessun altro, ma a cui possano fare riferimento molte parti diverse del codice. La variabile `logger` è un bell'esempio: è un servizio condiviso utilizzato da tutti i plug-in e dal server stesso. Tutti noi dobbiamo accedere a un oggetto di logging, e vogliamo che questo si trovi in una variabile che non sparisca. Diversamente da quanto accade con una variabile locale, vogliamo che sia visibile per tutte le funzioni nel plug-in e che rimanga disponibile

finché il plug-in è in uso.

Ricapitoliamo.

- Un blocco di codice è racchiuso tra i caratteri `{` e `}`.
- Le variabili dichiarate nel corpo di un blocco di codice sono locali, cioè valgono solo per quel blocco di codice. Non sono più disponibili quando la funzione termina e restituisce qualcosa.
- I parametri di una funzione sono locali per quella funzione.
- Le variabili dichiarate come `static` sopravvivono alle variabili locali dichiarate all'interno delle funzioni.

Di seguito trovi un breve plug-in che fornisce un comando `/caketower`. L'idea è quella di costruire una torre di blocchi di torta. Il codice presenta però un piccolo problema che riguarda le variabili locali e globali, e la torre potrebbe non risultare come te l'aspetti. Vediamo.

Plug-in: CakeTower

CakeTower/src/caketower/CakeTower.java

```
  package caketower;

  import net.canarymod.plugin.Plugin;
  import net.canarymod.logger.Logman;
  import net.canarymod.Canary;
  import net.canarymod.commandsys.*;
  import net.canarymod.chat.MessageReceiver;
  import net.canarymod.api.entity.living.humanoid.Player;
  import net.canarymod.api.world.position.Location;
  import net.canarymod.api.world.blocks.BlockType;
  import com.pragprog.ahmine.ez.EZPlugin;

  public class CakeTower extends EZPlugin {

1 public static int cakeHeight = 100;

  @Command(aliases = { "caketower" },
            description = "Build a tall tower of cakes",
            permissions = { "" },
            toolTip = "/caketower")
  public void cakeTowerCommand(MessageReceiver caller, String[] parameters) {
    if (caller instanceof Player) {
      Player me = (Player)caller;

      me.chat("1) cake height is " + cakeHeight); // Visualizza
      cakeHeight = 50;

2     int cakeHeight;
      cakeHeight = 5;
      me.chat("2) cake height is " + cakeHeight); // Visualizza

      makeCakes(me);

    }

  }

    public void makeCakes(Player me) {
      me.chat("3) cake height is " + cakeHeight);  // Visualizza
      Location loc = me.getLocation();
      loc.setY(loc.getY() + 2);
      setBlockAt(loc, BlockType.Stone);
      for(int i = 0;i < cakeHeight;i++) {
        loc.setY(loc.getY() + 1);
        setBlockAt(loc, BlockType.Cake);
      }
    }
  }
```

Quando questo codice viene eseguito, mostra a video il valore di `cakeHeight` (cioè l'altezza della torre) per tre volte. Nota che ci sono due dichiarazioni della variabile `cakeHeight`, una al punto **1** e una al punto **2**.

Cosa visualizzerà questo codice e quante torte finiranno nella torre? Prova a dedurlo e calcolarlo da solo, prima. Poi compila il codice e installalo usando

`build.sh` come sempre.

Cos'è successo?

Benvenuto del mondo meravigliosamente confuso del *shadowing*. Il shadowing è una regola di visibilità secondo cui una variabile può "fare ombra", letteralmente, a un'altra.

In questo codice, all'inizio del plug-in, viene dichiarata una variabile chiamata `cakeHeight`. È la variabile a cui ti aspetteresti di accedere da qualsiasi punto del plug-in, partendo da quella riga fino alla corrispondente parentesi graffa di chiusura (`}`).

Alla riga indicata con il punto **2**, però, troviamo dichiarata un'altra variabile con lo stesso identico nome.

Da qui fino alla prossima `}`, tutte le volte che citeremo `cakeHeight` intenderemo questa variabile locale, non quella a livello di classe. La versione locale "nasconde" la versione globale. Se la impostiamo a 5 e la visualizziamo, modifichiamo questa versione locale.

La chiamata a `makeCakes` usa poi la versione a livello di classe per costruire la torre. La funzione `makeCakes` non sa della variabile nascosta all'interno della funzione `cakeTowerCommand`, quindi finiamo con l'ottenere 50 blocchi; non 100, né cinque.

Morale della favola: non farlo. Come hai visto, i nomi delle variabili nascoste possono creare parecchia confusione. Assegna alle tue variabili dei nomi unici e facili da ricordare.

Prova da solo

Poiché la torre di torte non funziona ancora come dovrebbe, dovremo sistemarla.

Modifica `code/CakeTower/src/caketower/CakeTower.java` in modo da utilizzare solo la variabile locale `cakeHeight` (non quella a livello di classe) e passala a `makeCakes`.

Ora che sai dove risiedono le variabili e come utilizzarle, vediamo un paio di modi in cui puoi impiegare le pile di dati attraverso le collezioni di dati Java.

Usare un array Java

Per quanto le variabili con valori singoli siano utili e diffuse, può capitare che ti serva qualcosa di più. Magari vuoi tenere traccia di tutti i giocatori nel sistema, degli item nell'inventario di un giocatore, di un elenco di cose da fare, della lista della spesa o dei compiti che ti hanno dato a scuola.

Java ti fornisce diversi modi per tenere traccia e accedere alle pile di dati. Ne vedremo alcuni: il semplice `Array`, l'`ArrayList` e il comodissimo – per quanto a volte astruso – `HashMap` (trattato nel prossimo capitolo). Partiamo dall'array.

In Java non sono moltissime le collezioni di dati del genere array: troviamo i tipi `Array`, `Vector`, `LinkedList` e `ArrayList`. Ciascuno ha un proprio funzionamento interno, viene salvato in modo leggermente diverso e lavora in modo specifico con set di dati grandi e piccoli, ma l'idea di base resta la stessa per tutti.

Gli array sono probabilmente la forma più semplice di pile di dati. Li userai soprattutto quando hai piccoli elenchi di valori che vuoi creare direttamente nel codice e utilizzare subito. Gli array sono *a lunghezza fissa*, cioè non puoi aumentarne la dimensione né ridurla, per cui non sono la scelta ideale se devi aggiungere, eliminare o spostare spesso quanto contengono (per fare questo è meglio un `ArrayList`, come vedremo tra poco).

Puoi usare un array quando vuoi accedere ai suoi valori in base a un indice, o se vuoi percorrere tutti i valori e farci qualcosa con un ciclo `for`.

Puoi dichiarare un array racchiudendolo tra parentesi quadre (`[ ]`) e riempirlo di valori usando le parentesi graffe (`{ }`). Ecco un esempio di una lista di stringhe:

```
String[] grades = {"A", "B", "C", "D", "F", "Inc"};
```

Puoi accedere ai singoli elementi della lista usando le parentesi quadre:

```
String yourGrade = grades[2];
```

In questo caso, `yourGrade` (cioè "il tuo voto") sarà una C. Ehi, un momento: perché una C e non una B? Perché Java, come il linguaggio C e i suoi predecessori, inizia a contare da 0. Il primo elemento in una lista è 0, il secondo è 1, il terzo è 2 e così via. Ci farai l'abitudine. È come se per accedere al primo elemento dovessi aggiungere 0 all'inizio della lista, per accedere al secondo elemento dovessi aggiungere 1 e via di seguito.

Questo è esattamente il modo in cui un array viene salvato nella memoria del

computer: un gruppo di valori tutti su una riga. Poiché il primo elemento dell'array è proprio all'inizio della memoria, non c'è scarto. Il secondo valore è a uno dall'inizio, il terzo a due e così via.

Puoi capire quanto è lungo un array osservando il campo `length` (nota che questa non è una chiamata di funzione; non ci sono parentesi):

```
int numGrades = grades.length;
```

`numGrades` verrà impostato a 6, quindi i sei valori verranno numerati da 0 a 5. L'indice dell'ultimo elemento è sempre `length-1` (in questo caso 6 - 1, o 5).

Invece di inglobare tutti i valori nel codice come abbiamo fatto adesso, puoi creare un array di dimensione fissa e poi riempirlo con dei valori. Ecco come appare un array nel caso di una lista di valori `int`:

```
int[] quizScores = new int[5];
quizScores[0] = 85;
quizScores[1] = 92;
quizScores[2] = 63;
```

Sebbene ci sia spazio per 5 valori, qui abbiamo usato solo i primi 3, e tutto funziona.

Ottenere dei valori è come inserirli, solo che lo fai nell'altra direzione; usando il codice appena visto, puoi recuperare i valori in questo modo:

```
int myBestQuiz = quizScores[1];
int aBadDay = quizScores[2];
```

Per ottenere tutti i valori, puoi usare il caro vecchio ciclo `for`:

```
for (int i=0; i < 5; i++) {
    me.chat("Quiz score #" + i + ": " + quizScores[i]);
}
```

Ricorda che gli array hanno una dimensione fissa: se cerchi di recuperare un valore che va oltre la fine dell'array (come `quizScores[15]`), il plug-in genera un errore e si blocca. Nel nostro caso, poiché abbiamo definito che l'array `quizScores` deve avere una dimensione pari a 5, potremo salvare e recuperare i valori 0, 1, 2, 3 e 4. Ecco perché a metà ho usato `i < 5` (cioè minore di 5) invece di `<=`.

È molto più sicuro usare il campo `.length` di un array piuttosto che immettere direttamente "5", per esempio. È quindi preferibile scrivere il ciclo così:

```
for (int i=0; i < quizScores.length; i++) {
    me.chat("Quiz score #" + i + ": " + quizScores[i]);
}
```

Tra poco imparerai un modo ancora migliore per scorrere i valori in un array.

Adesso facciamo lo stesso, ma con i blocchi di Minecraft.

Ecco un esempio di codice per un plug-in veloce che costruisce una torre con blocchi di tipo diverso. Qui userò la funzione helper `setBlockAt` per modificare il materiale aria con uno degli altri numerosi materiali disponibili (i codici per i vari materiali si trovano nella documentazione dell'API Canary all'indirizzo http://bit.ly/1D28mLn). Vediamo cosa succede.

Plug-in: ArrayOfBlocks

ArrayOfBlocks/src/arrayofblocks/ArrayOfBlocks.java

```
package arrayofblocks;

import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import com.pragprog.ahmine.ez.EZPlugin;

public class ArrayOfBlocks extends EZPlugin {

  public void buildTower(Player me) {
    Location loc = me.getLocation();
    loc.setX(loc.getX() + 1); // Sopra il giocatore non va bene

    BlockType[] towerMaterials = new BlockType[5];

    towerMaterials[0] = BlockType.Stone;
    towerMaterials[1] = BlockType.Cake;
    towerMaterials[2] = BlockType.OakWood;
    towerMaterials[3] = BlockType.Glass;
    towerMaterials[4] = BlockType.Anvil;

    for (int i=0; i < towerMaterials.length; i++) {
      loc.setY(loc.getY() + 1); // Cresce di uno ogni volta
      setBlockAt(loc, towerMaterials[i]);
    }
  }

  @Command(aliases = { "arrayofblocks" },
           description = "Create an array of blocks",
           permissions = { "" },
           toolTip = "/arrayofblocks")
  public void arrayofblocksCommand(MessageReceiver  caller, String[] args) {
    if (caller instanceof Player) {
      Player me = (Player)caller;
      buildTower(me);
    }
  }
}
```

Installa `ArrayOfBlocks` con `build.sh`, ferma il server e riavvialo, quindi prova il comando `/arrayofblocks`. Dovresti vedere qualcosa di simile alla Figura 7.1.

Nota che ho posto il cuore del comando nella sua funzione, `buildTower`, invece che direttamente nella funzione `arrayofblocksCommand`.

Questo è un array semplice di lunghezza pari a 5 che riempiamo con un valore alla volta. Il ciclo `for` parte dall'indice 0 fino a 4 e trasforma il blocco con il nuovo materiale nell'elenco.

Figura 7.1 Array di blocchi.

Prova da solo

Ora è il tuo turno. Farai una piccola modifica per invertire l'ordine degli elementi della torre, così che l'incudine si trovi sul fondo e la pietra in cima.

Per farlo, devi modificare il ciclo `for`. Invece di andare da 0 a < 5, modifica il ciclo in modo che vada da 4 a >= 0. Ti do un suggerimento: in casi come questo, una sottrazione potrebbe funzionare meglio di un'addizione.

Ricompila, ferma il server e riavvialo, quindi prova nuovamente il comando `/arrayofblocks`.

Ora sai come lavorare con i cicli `for` e gli indici, ma a essere onesti, per Java questa è roba un po' vecchia. Gli array sono comodi, ma forse hai una scelta migliore.

Usare un ArrayList Java

In Java, un `ArrayList` tiene traccia di una lista di valori, proprio come un semplice array. Sono un po' difficili da dichiarare, ma sufficientemente facili da usare, oltre che più flessibili e sicuri dei vecchi oggetti `Array`. Negli `ArrayList` puoi fare aggiunte ed eliminazioni quante volte vuoi; non hanno una lunghezza fissa e possono crescere o ridursi secondo necessità.

Ecco un esempio di un `ArrayList` che contiene degli oggetti `Player`:

```
List<Player> myPlayerList = new ArrayList<Player>();
```

Quest'unica riga contiene molta più roba di quanta ne abbiamo vista finora. Per cominciare, ho chiamato la nuova lista `myPlayerList`. Osserva l'insolita sintassi con le parentesi angolari `<` e `>`. Devi specificare il tipo di lista che intendi creare (per due volte, in realtà). Java 7 fa un miglioramento: non devi ripetere il tipo sulla parte destra e puoi scrivere `List<Player> whatever = new ArrayList<>()`. Una piccola stranezza: nota che sebbene a sinistra ci sia `List`, `ArrayList` si trova a destra. È tipico di Java: la ragione è che `List` è il genitore e `ArrayList` un figlio specifico (per saperne di più, vai all'indirizzo http://theatln.tc/1xrdEJG). Andiamo avanti.

Con una lista puoi fare un sacco di cose divertenti, come aggiungere e rimuovere valori, recuperarli e verificare se un valore esiste. Ecco un esempio di codice.

ListPlay/src/listplay/ListPlay.java

```
  public void listDemo(Player me) {
1 List<String> listOfStrings = new ArrayList<String>();
2 listOfStrings.add("This");
  listOfStrings.add("is");
  listOfStrings.add("a");
  listOfStrings.add("list.");

3 String third = listOfStrings.get(2);
  me.chat("The third element is " + third);

4 me.chat("List contains " + listOfStrings.size() + " elements.");

5 listOfStrings.add(3, "fancy");

6 boolean hasIt = listOfStrings.contains("is");
  me.chat("Does list contain the word 'is'? " + hasIt);

  hasIt = listOfStrings.contains("kerfluffle");
  me.chat("Does the list contain the word 'kerfluffle'? " + hasIt);
  // Visualizza ciascun valore nella lista
  for(String value : listOfStrings) {
  me.chat(value);
```

```
}

7 listOfStrings.clear();
  me.chat("Now it's cleared out, size is " + listOfStrings.size());

  hasIt = listOfStrings.contains("is");
  me.chat("List contains the word 'is' now is " + hasIt);
  }
```

Prova da solo

Continua a leggere e cerca di capire cosa accade durante l'esecuzione. Ti sembra che abbia senso? Analizziamo il codice.

Per iniziare, al punto **1** troviamo `new`, che crea un elenco che conterrà delle `String`. Finora tutto bene. Al punto **2** aggiungiamo un paio di stringhe all'elenco, una alla volta, usando la funzione `add()`. Dopodiché possiamo provare a ottenere dei dati.

Al punto **3** otteniamo il terzo elemento nell'elenco, chiedendo l'indice 2. Il conteggio funziona come negli array, cioè parte da 0. Il terzo elemento è la stringa `"a"`.

Adesso verifichiamo quanti elementi sono contenuti nell'elenco usando la funzione `size()` al punto **4**: ci sono quattro elementi.

Uno dei vantaggi di un `ArrayList` rispetto a un array è che puoi aggiungere ed eliminare facilmente dei valori, anche in mezzo alla lista, come accade qui al punto **5**, dove usiamo `add()` e passiamo un indice 3. In questo modo aggiungiamo l'indice 3 alla lista e spostiamo tutti gli altri valori di una posizione verso il basso.

Puoi rimuovere dei valori, leggerli o fare altro, come vuoi. Non importa come aggiungi i valori o li mescoli; puoi cercare un dato valore in una lista senza doverla scorrere tutta, come avviene al punto **6** con la chiamata a `contains`, che cerca la parola `"is"`. Eccola lì. Forte! Puoi anche provare a cercare un valore che invece non c'è; per esempio, se cerchi il valore `"kerfluffle"`, `contains` restituirà `false`.

Hai visto come ottenere un singolo valore per indice impiegando la funzione `get`; ma se volessi percorrere i valori nella lista uno per uno?

Potresti ricorrere a un ciclo `for` come abbiamo fatto con gli array, ma è una tecnica fuori moda e a rischio di errore.

In alternativa, puoi usare un costrutto `for-each`. L’istruzione `for(String value : listOfStrings)` agisce come un ciclo `for` che si ripete lungo la collezione `listOfStrings`, impostando nel corpo del ciclo la variabile `value` per ogni elemento della lista man mano che procede. L’abbiamo visto all’opera la prima volta in `SkyCmd`.

Però! Poche righe di codice ma una lunga spiegazione. Ma l’idea è potente, e tra poco l’applicheremo a un plug-in.

Per finire, cosa succede quando ripuliamo completamente la lista (punto **7**)? Niente di che, visto che adesso è vuota.

Plug-in: ArrayAddMoreBlocks

Giochiamo un po'. Partiremo dal plug-in `ArrayOfBlocks`, ma lo modificheremo in modo che utilizzi un `ArrayList` invece di un semplice array. Così modificato, lo chiameremo `ArrayAddMoreBlocks`.

Poiché fare le aggiunte all'`ArrayList` è semplice, rendiamolo statico:

```
public static List<BlockType> towerMaterials = new ArrayList<BlockType>();
```

Ora, grazie alle meraviglie dell'`ArrayList`, possiamo aggiungere un paio di blocchi alla nuova torre ogni volta che chiamiamo `/arrayaddmoreblocks`:

ArrayAddMoreBlocks/src/arrayaddmoreblocks/ArrayAddMoreBlocks.java

```
  public void buildTower(Player me) {
    if (towerLoc == null) {
1     towerLoc = new Location(me.getLocation());
      towerLoc.setX(towerLoc.getX() + 2); // Sopra il giocatore non va bene
2     towerBase = new Location(towerLoc);
    }

    towerMaterials.add(BlockType.Glass);
    towerMaterials.add(BlockType.Stone);
    towerMaterials.add(BlockType.OakWood);

    for (BlockType material : towerMaterials) {
      logger.info("Building block at " + printLoc(towerLoc));
      setBlockAt(towerLoc, material);
      towerLoc.setY(towerLoc.getY() + 1); // Cresce di uno ogni volta
    }
  }
```

Per reimpostare la lista, si usa la funzione `clear()`; nello specifico, ho impostato un comando distinto solo per fare questo.

ArrayAddMoreBlocks/src/arrayaddmoreblocks/ArrayAddMoreBlocks.java

```
public void clearTower() {
  if (towerLoc == null) {
    return;
  }
  while (towerBase.getY() < towerLoc.getY()) {
    setBlockAt(towerBase, BlockType.Air);
    logger.info("Clearing block at " + printLoc(towerBase));
    towerBase.setY(towerBase.getY() + 1); // Cresce di uno ogni volta
  }
  towerLoc = null; // Reimposta per la prossima torre
  towerBase = null;
  towerMaterials.clear();
}
```

Un'ultima annotazione interessante sulle variabili: osserva che ai punti **1** e **2** ho usato `new` per creare una nuova variabile, come in `towerBase = new Location(towerLoc)`, invece di assegnarla, come in `towerBase = towerLoc`. Perché ho agito così?

Ricorda che quando crei una variabile con `new`, la variabile vera e propria risiede in qualche punto della memoria, e che il nome che le hai dato è solo un nome. Se avessi digitato `towerBase = towerLoc`, ci sarebbe stata solo una `Location` con due nomi, e non due variabili distinte. Digitando `towerBase = new Location(towerLoc)` ho creato una variabile completamente nuova chiamata `towerBase` che ha copiato i valori da `towerLoc`.

Se intendi modificare i valori di una variabile ma non vuoi alterare l'originale, crea sempre una copia.

Il plug-in completo è in *code/ArrayAddMoreBlocks*. Compila il codice, ferma il server e riavvialo, quindi prova il plug-in, costruendo ed eliminando alcune torri. Ricorda che potresti dover ruotare e puntare verso un'altra direzione per vedere la torre.

Prova da solo

Ora modifica il codice e aggiungi un paio di materiali da costruzione diversi. Prova prima con un solo materiale, per esempio `Dirt` (terra), oppure realizza una bella miscela di `Dirt` e `Grass` (terra ed erba). Decidi tu.

Per continuare

In questo capitolo hai imparato quali sono le differenze tra variabili locali e globali. Puoi usare un semplice array oppure un `ArrayList` più flessibile per contenere una pila di dati e percorrerla usando un iteratore `for-each`.

Gli array sono eccezionali se non ti interessa trovare uno degli oggetti che contengono. Se invece vuoi che questi siamo rintracciabili per nome – come un `Player` –, allora ti serve qualcosa di più articolato.

Nel prossimo capitolo spiegheremo come utilizzare le `HashMap` per salvare i dati per nome (o per `Location` o secondo qualsiasi altro criterio ti occorra).

La tua toolbox

44%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.
- Usare gli oggetti Java.
- Usare le istruzioni `import` per i package Java.
- Usare `new` per creare gli oggetti.
- Aggiungere un nuovo comando a un plug-in.
- Lavorare con gli oggetti `Location`.
- Trovare blocchi ed entità.
- Usare le variabili locali.
- Usare le variabili globali a livello di classe.
- Usare gli `ArrayList`.

Usare pile di variabili: le HashMap

"HashMap" è un nome bizzarro. La parte "hash" si riferisce al suo funzionamento interno, e non ha nulla a che fare con il modo in cui la si utilizza. Quella che più ci interessa, invece, è la parte che "mappa" una chiave – che può essere qualsiasi cosa – su un valore, anche questo di qualsiasi tipo.

Altri linguaggi parlano di *dizionario*, *array associativo* o *memoria associativa*: sono tutti termini che indicano la stessa cosa. Con un array, usi un intero come indice; con un'`HashMap`, puoi usare un oggetto qualsiasi come chiave, in particolare oggetti `String` (vedi la Figura 8.1).

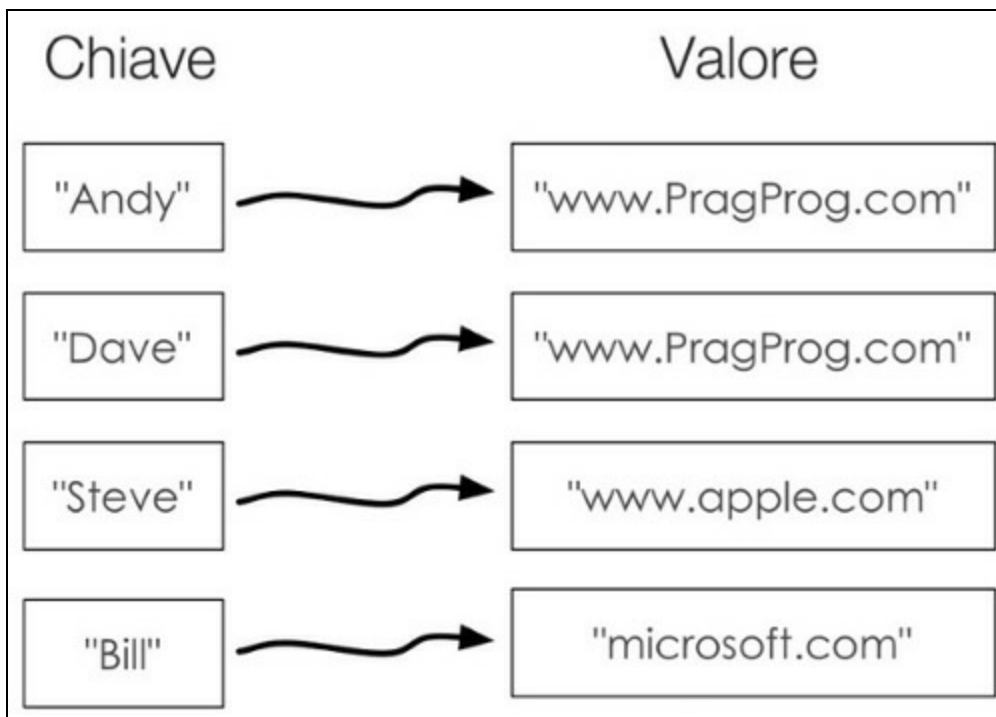



Figura 8.1 Un'HashMap.

Usare un’HashMap di Java

Di `HashMap` ne useremo tante nei plug-in, perché rappresentano un ottimo sistema per tenere traccia dei giocatori, delle mucche che hai creato o di quant’altro desideri. Per quanto un’`HashMap` lavori come un array, non la usi nello stesso modo. Quando abbiamo studiato gli array abbiamo visto che si può impostare un valore con un indice, così:

```
myList[9] = “Andy”;
String who = myList[9];
```

Con un’`HashMap` non puoi ricorrere a questo tipo di assegnazione e di notazione con le parentesi quadre. Devi invece utilizzare le sue funzioni `put` e `get`. Il vantaggio è che puoi impiegare qualsiasi cosa come chiave, anche se in genere utilizzerai una `String`. Immagina di avere un’`HashMap` chiamata `myHash`:

```
myHash.put(“Andy”, “www.PragProg.com”);
// Funziona come myHash[“Andy”] = “www.PragProg.com”;

String myUrl = myHash.get(“Andy”);
// Funziona come myUrl = myHash[“Andy”];
```

Ora `myUrl` avrà il valore `“www.PragProg.com”`.

Le `HashMap` sono perfette per memorizzare molti dati indicizzati da una stringa, proprio come un dizionario. Cerchi una parola e ottieni una serie di dati. Un’`HashMap` può essere utilizzata anche per contenere un insieme di proprietà con nome per un oggetto, e la cosa interessante è che i nomi delle proprietà sono proprio stringhe. Puoi aggiungere nuove proprietà o eliminarne mentre il plug-in è in esecuzione, qualcosa che è difficile fare con le funzioni immesse direttamente o con i nomi delle variabili.

Quello che segue è un piccolo programma indipendente nel quale puoi fare qualche esperimento con un’`HashMap`. Immagina di creare un plug-in che implementa un ambiente del genere *Hunger Games*, in cui vuoi avere un punteggio per ogni giocatore. Puoi tenere traccia di ogni punteggio servendoti di un’`HashMap`.

HashPlay/src/hashplay/HashPlay.java

```
1 HashMap<String, Integer> currentScores = new HashMap<String, Integer>();

  public static void addToScore(HashMap<String, Integer> allScores,
                      String playerName,
2                     int amount) {
3   int score = allScores.get(playerName);
4   score += amount;
```

```
5   allScores.put(playerName, score);
 }
```

Ecco come utilizzarla.

HashPlay/src/hashplay/HashPlay.java

```
currentScores.put(“Andy”, 1001);
currentScores.put(“Bob”, 20);
currentScores.put(“Carol”, 50);
currentScores.put(“Alice”, 896);
addToScore(currentScores, “Bob”, 500);
me.chat(“Bob’s score is “ + currentScores.get(“Bob”));
```

Compilando ed eseguendo questo codice in un plug-in, otterrai la risposta prevista di 520 per `Bob`. Analizziamo il codice un pezzo alla volta.

Al punto **1** trovi la chiamata a `new()` che crea una nuova `HashMap`. Poiché l’`HashMap` deve sapere cosa utilizzeremo come chiave e come valore, dovremo passare i nomi dei tipi racchiusi tra le parentesi angolari (`<` e `>`), proprio come abbiamo fatto per `ArrayList`; la differenza è che qui dobbiamo passare due tipi, uno per la chiave (che è una `String`) e uno per il valore (un `Integer`). (Perché `Integer` e non `int`? Perché è Java. Quando si usano collezioni come `HashMap` e `ArrayList`, devi fare riferimento ai tipi primitivi – `int`, `float` – con i loro nomi di classe, `Integer` e `Float`. Il magico autoboxing si occupa di eseguire la conversione da `Integer` a `int`. Ricorda di scrivere i nomi completi con l’iniziale maiuscola tra le parentesi angolari e tutto filerà liscio.)

Ottenuta una nuova `HashMap` chiamata `currentScores`, possiamo procedere e creare alcune voci di prova nell’hash con i nomi dei giocatori e i loro punteggi. Questa volta, però, faremo qualcosa di leggermente diverso.

Creeremo una nuova funzione helper che incrementerà il punteggio. Puoi vedere l’inizio al punto **2**. Si tratta di una semplice funzione che esegue tre azioni. Quando ti trovi a dover compiere una serie di passaggi per più di una volta, non serve che li scrivi e poi li copi e incolli. Crea invece una funzione che fa tutto questo per te, e poi chiamala quando ti serve.

Dichiariamo la nostra funzione `addToScore()` in modo che prenda tre argomenti: l’`HashMap` di tutti i punteggi, la stringa con il nome del giocatore e il valore intero da aggiungere al punteggio. Con questi dati a disposizione, svolgiamo tre azioni.

1. Otteniamo il punteggio corrente del giocatore il cui nome si trova in `playerName` (sulla riga al punto **3**).

2. Aumentiamo il punteggio dell'ammontare che abbiamo passato al punto **4**.
3. Salviamo il valore appena incrementato nell'hash al punto **5**.

Nota che non stiamo restituendo alcun valore specifico per questa funzione, poiché è dichiarata come `void`, esattamente com'è `main`. Quello che avviene è la modifica dell'`HashMap` passata, in questo caso la variabile globale `currentScores`.

Tuttavia, la maggior parte delle volte ci serve che le funzioni restituiscano un valore utile. Per esempio, la prossima funzione aggiunge 10 a qualsiasi valore che viene passato:

```
public static int addTen(int originalNumber) {
    int newNumber = originalNumber + 10;
    return newNumber;
}
```

Quello che abbiamo fatto di diverso è specificare il tipo di valore che vogliamo venga restituito (in questo caso `int`) invece di usare `void` nella dichiarazione. Dopodiché usiamo la parola chiave `return` con il valore che vogliamo restituire al chiamante. In genere chiamerai `return` come ultimissima cosa nella funzione; questo perché specifica quale valore restituire e lo farà immediatamente. Dopo la chiamata a `return`, nel metodo non verrà eseguito altro codice, e avrai terminato.

Prova da solo

Modifica il file sorgente `HashPlay.java` in modo che il punteggio di nessun giocatore possa scendere sotto lo 0 o superare 1.000. Usa una funzione helper che restituisca un valore compreso tra 0 e 1.000.

Trovi la mia soluzione nel file `HashPlayClamp/src/hashplay/HashPlay.java`. Qui scoprirai anche un altro trucchetto: invece di rendere la funzione helper `public`, l'ho dichiarata `private`. Perché?

Mantenere le cose private o renderle pubbliche

Finora, quando abbiamo creato variabili statiche e nella definizione di funzioni e plug-in, abbiamo privilegiato l'uso della parola chiave `public`. Questa parola chiave dice a Java che quanto stiamo definendo dovrà essere accessibile pubblicamente, ovvero potrà essere usato da tutto il codice di questo plug-in oltre che dagli eventuali altri plug-in nel sistema (come con `logger`).

C'è però un'alternativa. Puoi creare funzioni, variabili o anche oggetti helper che *nessun altro* può vedere. Possono cioè essere `private`.

In programmazione esiste una regola elementare: non esporre quello che vuoi che resti privato.

In altre parole, se usi una funzione o qualcosa che *solo* tu dovresti usare, allora contrassegnalo come privato, così che nessun altro dall'esterno possa utilizzarlo. Ma perché farlo?

Immagina che il tuo plug-in contenga una funzione che identifica un giocatore come `SuperUberHighWizard` (più o meno "super-arci-strega di alto livello"), e di non volere che altri plug-in chiamino questa funzione su un giocatore a loro scelta.

Là dove dichiareresti la funzione nel plug-in come `public` (tralasciando per ora altre parti del codice), così:

```
public class WizardingWorld extends EZPlugin {
    ...
    public void makeSuperUberHighWizard(Player p) {
        ...
    }
    // Qualsiasi plug-in può chiamare WizardingWorld.makeSuperUberHighWizard()
}
```

puoi renderla `private`, così:

```
public class WizardingWorld extends EZPlugin {
    ...
    private void makeSuperUberHighWizard(Player p) {
        ...
    }
    // Solo le funzioni in questo plug-in possono chiamare SuperUberHighWizard()
}
```

STACK E HEAP

Potresti aver notato che a volte creiamo e assegniamo un valore a una variabile direttamente, come in `int a = 25`, mentre altre volte per creare un oggetto come un plug-in utilizziamo la parola chiave `new`.

Questi due tipi di oggetti vengono salvati in modo diverso. I *valori immediati* come gli interi e i decimali sono conservati in un elenco di chiamate alla funzioni Java da te creato. Questo elenco si chiama *stack*, cioè pila. È come una pila di libri: ogni nuova chiamata a una funzione genera un nuovo libro nello stack, e dopo la sua esecuzione viene rimossa e tu vieni riportato al libro precedente. Al termine della funzione, il "libro" viene eliminato, e con esso le sue variabili locali.
Gli oggetti creati con `new` vengono invece salvati in uno spazio di memoria più grande chiamato *heap*, che vuol dire "mucchio", e puoi averli a disposizione anche dopo che una funzione è terminata (come accade con i plug-in).
Quello che deve esserci è semplicemente una variabile da qualche parte che punta al tuo oggetto; può essere una variabile locale e passata oppure una variabile globale. Java tiene traccia di quante variabili fanno riferimento all'oggetto creato con `new`. Una volta che l'oggetto non è più utilizzato, viene gettato nel cestino. Quando il sistema lo rileva, svuota e il cestino, e l'oggetto sparisce per sempre (Java chiama questo meccanismo *garbage collection*, cioè raccolta dei rifiuti). Come abbiamo accennato verso la fine del Capitolo 7, è necessario prestare molta attenzione: l'assegnazione fornisce due riferimenti allo stesso oggetto; `new` crea invece una copia nuova di un oggetto.

In generale, se intendi creare una funzione che utilizzerai solo in un dato plug-in e che gli altri plug-in non potranno chiamare direttamente, allora rendila `private`. Lo stesso vale per le variabili. Se invece vuoi che altri plug-in la vedano e la usino, allora rendila `public`. Per i nostri esempi, partiremo con una funzione helper `private`.

Plug-in: NamedSigns

Raccogliamo un po' le idee e creiamo un plug-in che utilizza funzioni helper (con e senza la restituzione di valori), funzioni e variabili `private`, un accesso di basso livello a un array e un hash che memorizza nomi e posizioni.

Questo plug-in consente di creare dei cartelli (*sign*) nel gioco e di inserire del testo su uno qualsiasi dei cartelli in base al nome. Immagina di creare due cartelli, uno chiamato `one` e uno `two`, digitando questi comandi nella finestra della chat:

```
/signs new one
/signs new two
/signs set one Hello!
/signs set two Goodbye!
```

Vedrai qualcosa del genere:

Possiamo modificare il testo dei cartelli come vogliamo eseguendo un altro comando `signs set`, così:

```
/signs set two Adios!
```

Vediamo il plug-in per intero, poi lo analizzeremo nel dettaglio.

NamedSigns/src/namedsigns/NamedSigns.java

```
package namedsigns;
import java.util.HashMap;
import java.util.Iterator;
```

```
 import java.util.Map;
 import net.canarymod.plugin.Plugin;
 import net.canarymod.logger.Logman;
 import net.canarymod.Canary;
 import net.canarymod.commandsys.*;
 import net.canarymod.chat.MessageReceiver;
 import net.canarymod.api.entity.living.humanoid.Player;
 import net.canarymod.api.world.World;
 import net.canarymod.api.world.position.Location;
 import net.canarymod.api.world.blocks.Block;
 import net.canarymod.api.world.blocks.BlockType;
 import net.canarymod.api.world.blocks.Sign;
 import com.pragprog.ahmine.ez.EZPlugin;

 public class NamedSigns extends EZPlugin {
1  private static Map<String,Location> signs = new HashMap<String,Location>();

2  private void usage(Player me) {
     me.chat("Usage: signs new name");
     me.chat("       signs set name message");
   }

   private void parseArgs(Player me, String [] args) {
3    if (args.length < 3) {
       usage(me);
       return;
     }
     if (args[1].equalsIgnoreCase("new"))  {
       makeNewSign(me, args);
     }
     if (args[1].equalsIgnoreCase("set"))  {
4      if (args.length < 4) {
         usage(me);
         return;
       }
       setSign(me, args);
     }
   }

   // nuovo sign_name dei cartelli
5  private void makeNewSign(Player me, String [] args) {
     Location loc = me.getLocation();
     loc.setX(loc.getX() + 1); // Sopra il giocatore non va bene
     int y = loc.getWorld().getHighestBlockAt((int)loc.getX(),(int)loc.getZ());
     loc.setY(y);
     signs.put(args[2], loc);
     setBlockAt(loc, BlockType.SignPost);
   }

   // Imposta il sign_name della riga 1
6  private void setSign(Player me, String [] args) {
     String name = args[2];
     String msg = args[3];
     if (!signs.containsKey(name)) {
       // Nessun cartello con questo nome
       me.chat("No sign named " + name);
       return;
   }
   Location loc = signs.get(name);
   World world = loc.getWorld();
7  Sign sign = (Sign)world.getTileEntity(world.getBlockAt(loc));
   sign.setTextOnLine(msg, 0);
   sign.update();
   }
```

```
  @Command(aliases = { "signs" },
           description = "Create and name signposts", permissions = { "" },
           toolTip = "/signs new name, or /signs set name message")
  public void signsCommand(MessageReceiver caller, String[] args) {
    if (caller instanceof Player) {
      Player me = (Player)caller;
      parseArgs(me, args);
    }
  }
}
```

Il codice è lungo, ma in parte è simile a qualcosa che abbiamo già visto. Consideriamo le sue parti principali.

- Un'`HashMap` statica privata al punto **1**.
- Una funzione helper privata che non restituisce nulla (`void`) al punto **2**. Questa funzione mostra un messaggio al giocatore. La chiamiamo solo se scopriamo che il giocatore non ha digitato correttamente il comando.
- Una funzione helper privata al punto **5**. Questa funzione crea nuovi cartelli.
- Una funzione helper privata al punto **6**. Questa funzione modifica il nome dei cartelli esistenti.
- Un controllo sulla lunghezza dell'array `args` nelle righe ai punti **3** e **4**. Poiché l'utente potrebbe non aver digitato abbastanza parole utili, dobbiamo verificare che la lunghezza sia sufficiente, altrimenti segnalare l'errore.
- Un tocco di magia al punto **7**. Secondo la documentazione di Canary, questo è il modo per ottenere un oggetto `Sign` da un `Block`.

Questo è il cuore. Vediamo ora i vari segmenti uno alla volta.

L'HashMap dei cartelli

Per iniziare, impostiamo un'`HashMap` chiamata `signs` per tenere traccia delle `Locations` (le posizioni) in base al nome (una `String`). Quando creiamo un cartello, inseriamo la sua posizione nell'hash usando il nome che l'utente ci ha fornito. Quando dovremo impostare il testo su un cartello, recupereremo la posizione del cartello dall'hash tramite il nome.

La funzione parseArgs

Quando l'utente digita un comando nel gioco, bisogna fare un paio di controlli.

Nota che qui usiamo un parametro della funzione del comando che non abbiamo usato prima, cioè `String[] args`. Si tratta degli argomenti che il giocatore digita insieme al comando nella finestra della chat. Per esempio, se digitiamo

```
/signs
```

questo verrà passato a `signsCommand` nell'array `args` come `args[0]`; `args` avrà una lunghezza pari a 1. Se digitiamo

```
/signs set one Hello!
```

allora `args[0]` sarà `"signs"`, `args[1]` sarà `"set"`, `args[2]` sarà `"one"` e `args[3] sarà "Hello!"`.

Sappiamo che nell'array `args` ci vogliono almeno tre valori, che saranno `"signs new name"` o `"signs set name something"`.

Ricorda che quando lavori con gli array devi controllare che questi siano lunghi abbastanza, altrimenti otterrai un messaggio d'errore dal server che termina con qualcosa del genere:

```
Caused by: java.lang.ArrayIndexOutOfBoundsException: 1
        at namedsigns.NamedSigns.makeSign(NamedSigns.java:26)
        at namedsigns.NamedSigns.signsCommand(NamedSigns.java:47)
        ... 15 more
```

Qui stiamo controllando la lunghezza degli argomenti per conto nostro, ma potremmo anche impostare la parte "`min == 3`" nell'annotazione `@Command`. In questo modo, se non ci sono almeno tre argomenti, il sistema invierà al giocatore un messaggio con il nostro `toolTip` (cioè il nostro suggerimento) come aiuto.

Certi di avere almeno tre valori nell'array `args` con cui lavorare, vediamo cos'ha digitato effettivamente il giocatore.

Il comando "/signs new"

Ecco la parte relativa al comando `"new"`:

NamedSigns/src/namedsigns/NamedSigns.java

```
// nuovo sign_name dei cartelli
private void makeNewSign(Player me, String [] args) {
  Location loc = me.getLocation();
  loc.setX(loc.getX() + 1); // Sopra il giocatore non va bene
  int y = loc.getWorld().getHighestBlockAt((int)loc.getX(),(int)loc.getZ());
  loc.setY(y);
  signs.put(args[2], loc);
setBlockAt(loc, BlockType.SignPost);
}
```

Per iniziare, prendiamo un blocco che ci è comodo accanto al giocatore (`getX()`

`+1`) e recuperiamo il blocco più alto in quel punto con `getHighestBlockAt()`. In questo modo non collocheremo il cartello sott'acqua, nella roccia o in altri posti non adatti.

Quindi salviamo la posizione di questo blocco nell'hash, usando il `name` che ci ha fornito il giocatore (`args[2]`).

Infine impostiamo il tipo di quel blocco a `BlockType.SignPost`. Ecco che è diventato un cartello.

Il comando "/signs set"

Ecco la parte relativa al comando `"set"`:

NamedSigns/src/namedsigns/NamedSigns.java

```
// imposta il sign_name della riga 1
private void setSign(Player me, String [] args) {
  String name = args[2];
  String msg = args[3];
  if (!signs.containsKey(name)) {
    // Nessun cartello con questo nome
    me.chat("No sign named " + name);
    return;
  }
  Location loc = signs.get(name);
  World world = loc.getWorld();
  Sign sign = (Sign)world.getTileEntity(world.getBlockAt(loc));
  sign.setTextOnLine(msg, 0);
  sign.update();
}
```

Nota che qui impostiamo due variabili locali, `name e msg`, per `args[2]` e `args[3]`. Perché preoccuparsene? Non sono la stessa cosa? Sì, lo sono, ma è molto più semplice leggere `name` invece di `args[2]` e cercare di ricordarsi che `2` è il nome e `3` il messaggio.

Ora controlleremo che nell'hash ci sia effettivamente una voce relativa al `name`; in caso contrario lo segnaleremo al giocatore. Per contro, possiamo facilmente ottenere la posizione del blocco del cartello.

E ora il tocco magico. Possiamo ottenere il blocco nel punto giusto, ma rimane sempre un blocco, un `net.canarymod.api.world.blocks.Block`. Un blocco non sa nulla delle funzioni di un `Sign`. Un `Sign` è un tipo di `Block`, quindi dovremo convincere Java a cavarne fuori un `net.canarymod.api.world.blocks.Sign`.

La documentazione di Canary dice di chiamare `world.getTileEntity()` sul blocco e

poi eseguire il cast usando l'operatore di cast `(Sign)` per creare un cartello vero e proprio. Con la variabile `sign` a disposizione, possiamo chiamare le due funzioni `Sign` che ci servono: `setTextOnLine()` e `update()`.

`setTextOnLine()` inserisce una riga di testo sul cartello all'indice dato (0 in questo caso), mentre `update()` garantisce che il cartello venga ridisegnato nel client così che sia possibile vedere il nuovo testo.

Può sembrare un codice molto esteso e complesso, ma se lo prendi un pezzettino alla volta, non è così difficile. Adesso, però, tocca a te fare qualche modifica.

Prova da solo

Ora come ora, il plug-in imposta solo la prima riga del cartello, ma su un cartello ci possono essere fino a quattro righe di testo. Modifica il plug-in in modo che se l'utente digita delle parole in più, passerai ogni parola a `sign.setTextOnLine()`. Ricorda: se il testo è di quattro righe, queste verranno numerate come 0, 1, 2 e 3. Hai già `setTextOnLine(msg, 0)`, quindi ti basta inserire il resto se `args` è sufficientemente lungo.

Per continuare

In questo capitolo abbiamo visto come usare un'`HashMap` per tenere traccia di dati importanti del gioco in base al nome o ad altri oggetti, e come rendere le funzioni `private` per impedire che altri plug-in vi accedano o `public` per consentirlo.

Nel prossimo capitolo faremo qualcosa di più che rispondere ai comandi dell'utente. Vedremo come ascoltare il server di Minecraft e reagire agli eventi del gioco man mano che si verificano, oltre che a crearne di personali.

La tua toolbox

50%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.
- Usare gli oggetti Java.
- Usare le istruzioni `import` per i package Java.
- Usare `new` per creare gli oggetti.
- Aggiungere un nuovo comando a un plug-in.
- Lavorare con gli oggetti `Location`.
- Trovare blocchi ed entità.
- Usare le variabili locali.
- Usare le variabili globali a livello di classe.
- Usare gli `ArrayList`.
- Usare le `HashMap`.
- Usare `private` e `public` per controllare la visibilità.

Modificare, generare e ascoltare in Minecraft

Quello che faremo nelle prossime pagine è andare oltre i semplici comandi visti finora (e oltre i calamari-bomba) e affrontare alcune azioni più articolate che puoi compiere in Minecraft. Alla fine di questo capitolo, sarai capace di influire sul comportamento del gioco addirittura senza eseguire neanche un comando.

Tutto quello che devi fare è imparare ad ascoltare per "conoscere" gli eventi di Minecraft. Ascolterai gli eventi, agirai su di essi e programmerai anche eventi personali che dovranno essere attivati in un momento futuro.

Dal codice del plug-in, puoi cambiare i blocchi e le entità già presenti nel gioco, così come puoi crearne di nuovi. In particolare, vedremo come compiere le seguenti azioni.

- Modificare i blocchi esistenti, ovvero cose come posizioni, proprietà e contenuti.
- Modificare le entità esistenti, per esempio intervenendo sulle proprietà di un `Player`.
- Generare nuove entità e nuovi blocchi.

Alcune di queste cose le abbiamo già fatte; per esempio abbiamo modificato la posizione di un `Player` e abbiamo generato alcuni calamari (`Squid`). Diamo un'occhiata più da vicino a quello che si può fare con gli elementi di base nel mondo di Minecraft; a seguire vedremo come rispondere agli eventi che accadono nel gioco per agire su di essi e per crearne di nuovi.

Modificare i blocchi

La ricetta di base di un oggetto blocco in Minecraft è riportata nella documentazione di Canary in `net.canarymod.api.world.blocks.Block` (`https://ci.visualillusionsent.net/job/CanaryLib/javadoc`).

Un blocco contiene numerose funzioni interessanti, ma non possiamo trattarle tutte. A seguire trovi però un elenco delle cose più utili e interessanti che puoi fare con un blocco.

- `getLocation()` restituisce la `Location` per quel blocco. In ciascuna posizione nel mondo può esistere un unico blocco, e ogni posizione contiene un blocco, anche se è fatto di aria.
- `getType()` restituisce il `BlockType` di cui è fatto quel blocco.
- `rightClick(Player player)` simula un clic con il tasto destro del mouse sul blocco. È utile per inserire nei blocchi modifiche come leve, pulsanti e porte.

Giochiamo con qualche blocco nello stile di Minecraft.

Plug-in: Stuck

Vediamo un plug-in che ti permette di imprigionare un giocatore nella roccia (il plug-in completo si trova in *code/Stuck*). Quando esegui il comando `stuck` con il nome di un giocatore, questi verrà rinchiuso in una pila di blocchi. (Se sei da solo sul server, il nome del giocatore potrebbe essere semplicemente “player”.)

Analizziamo il plug-in segmento per segmento e poi assembliamo il tutto.

Le parti più interessanti si trovano in una funzione helper distinta chiamata `stuck`. La sezione principale del plug-in dovrebbe ormai esserti familiare.

Stuck/src/stuck/Stuck.java

```
package stuck;
import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import com.pragprog.ahmine.ez.EZPlugin;

public class Stuck extends EZPlugin {

  @Command(aliases = { “stuck” },
            description = “Trap a player in cage of blocks”,
            permissions = { “” },
            min = 2,
            toolTip = “/stuck name”)
  public void stuckCommand(MessageReceiver caller, String[] args) {
  Player victim = Canary.getServer().getPlayer(args[1]);
  if (victim != null) {
  stuck(victim);
  }
}
```

Nella specifica `@Command`, impostiamo il numero minimo di argomenti a `2`. In questo modo non dovremo scrivere del codice per “autocontrollarci”; ci penserà automaticamente il sistema. In `stuckCommand`, proveremo a ottenere il nome di un certo giocatore, cosa che potrebbe anche non funzionare. Se così fosse (cioè se online non c’è nessun giocatore con quel nome), l’istruzione `if` fallirà e verremo riportati indietro senza far niente.

Se invece funziona (cioè se troviamo il giocatore), potremo procedere e chiamare `stuck`, passando l’oggetto giocatore che abbiamo recuperato dal server.

Ecco l’inizio della funzione `stuck`.

Stuck/src/stuck/Stuck.java

```
public void stuck(Player player) {
  Location loc = player.getLocation();
  int playerX = (int) loc.getX();
  int playerY = (int) loc.getY();
  int playerZ = (int) loc.getZ();
  loc.setX(playerX + 0.5); loc.setY(playerY); loc.setZ(playerZ + 0.5);
  player.teleportTo(loc);
```

La prima cosa da fare nella funzione `stuck` è determinare la posizione corrente del giocatore in `loc`. Nelle prossime righe, faremo in modo di teletrasportare il giocatore al centro del blocco in cui si trova in questo momento, così sarà più facile accumulargli intorno altri blocchi.

Come procedere? Sappiamo che un giocatore occupa fino a due blocchi. La sua posizione è il punto in cui si trovano effettivamente le sue gambe e i suoi piedi. Il blocco appena sopra (y+1) costituisce la sua testa e il suo torace. Ci serve quindi una pila di due blocchi per ciascuno dei quattro i lati del giocatore più un blocco sotto e uno sopra, per un totale di dieci blocchi, come mostrato nella Figura 9.1.

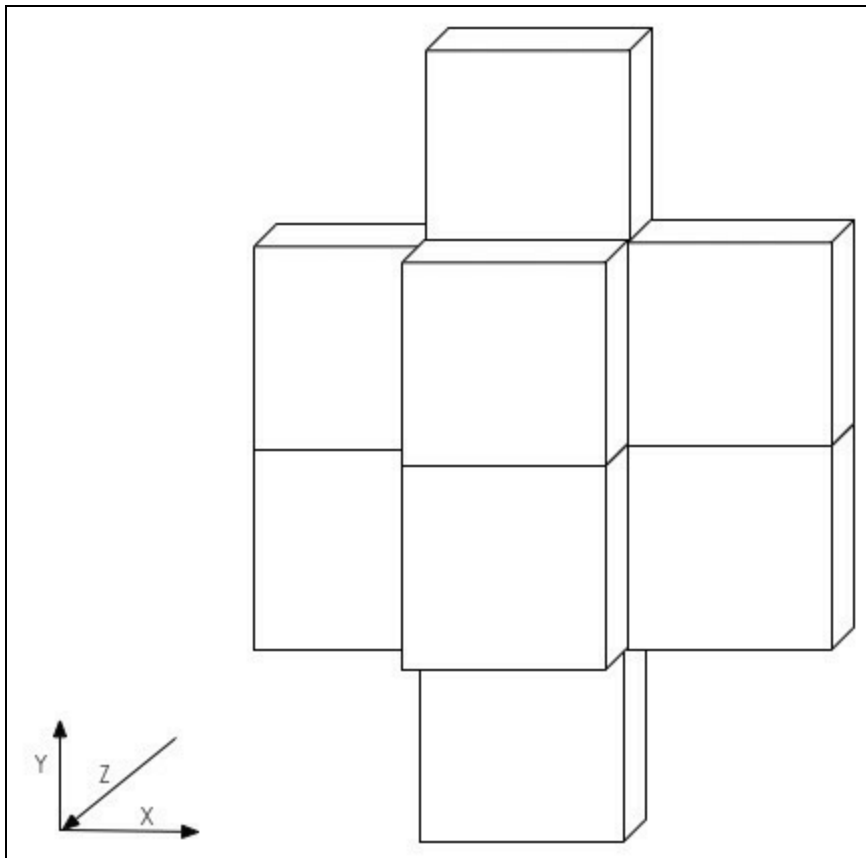



Figura 9.1 Intrappolare un giocatore nei blocchi.

Basandoci sulla posizione del giocatore, sappiamo dove va collocato ciascuno

di questi blocchi. Per esempio, potremmo aver bisogno di dieci set di coordinate, una per ogni *offset* (cioè distanza) dal blocco di base. Ci serve cioè una “lista di liste”.

È quello che vediamo qui: abbiamo un array `int` di dieci elementi, e ogni elemento è un array `int` di tre offset, uno per ciascuno dei valori x, y e z:

Stuck/src/stuck/Stuck.java

```
int[][] offsets = {
//x,   y,    z
 {0,   -1,   0},
 {0,   2,    0},
 {1,   0,    0},
 {1,   1,    0},
 {-1,  0,    0},
 {-1,  1,    0},
 {0,   0,    1},
 {0,   1,    1},
 {0,   0,   -1},
 {0,   1,   -1},
};
```

Per scorrere questa lista di liste, usiamo un ciclo `for`. Il primo elemento nella lista è indicizzato a 0, quindi procediamo verso l’alto (ma senza includerlo). Usando la lista degli offset per tutta la sua lunghezza invece di fissarci su un numero come 10, potremo aggiungere più facilmente dei blocchi supplementari ogni volta che vorremo (ricorda, stiamo aggiungendo gli offset `playerX`, `playerY` e `playerZ` dal codice precedente).

Stuck/src/stuck/Stuck.java

```
for(int i = 0; i < offsets.length; i++) {
  int x = offsets[i][0];
  int y = offsets[i][1];
  int z = offsets[i][2];
  setBlockAt(new Location(x+playerX, y+playerY, z+playerZ),
    BlockType.Stone);
}
```

Percorriamo così la lista degli offset. Per ogni indice della lista (che è in `i`), dobbiamo prendere i tre elementi x, y e z. In ciascuno dei piccoli array, x è il primo all’indice 0. La sintassi Java permette di lavorare con array di array scrivendo entrambi gli indici, mettendo davanti la lista principale. Prova a immaginare questo insieme di numeri come se fosse una tabella o una matrice, con righe e colonne, come quelle dei fogli di calcolo di Microsoft Excel. Puoi specificare gli indici secondo un ordine in cui la riga viene prima, seguita dalla colonna. Per ogni passaggio del ciclo, prendiamo i valori x, y e z dalla lista.

Quella è la posizione del blocco che andrà trasformato in pietra.

Otteniamo allora il blocco nella posizione che ci interessa, in questo caso aggiungendo gli offset x, y e z alla posizione del giocatore (`playerX`, `playerY` e `playerZ` nel codice). Preso il blocco, impostiamo il suo materiale a pietra usando la costante `BlockType.Stone`. I diversi tipi di blocco sono elencati nella documentazione in `net.canarymod.api.world.blocks.BlockType`. Per esempio, per rimuovere un blocco senza distruggerlo potresti impostare il suo materiale a `BlockType.Air` (aria), come abbiamo fatto in precedenza nelle torri di array.

Ecco il codice completo del plug-in.

Stuck/src/stuck/Stuck.java

```
package stuck;
import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import com.pragprog.ahmine.ez.EZPlugin;

public class Stuck extends EZPlugin {

  @Command(aliases = { "stuck" },
            description = "Trap a player in cage of blocks",
            permissions = { "" },
            min = 2,
            toolTip = "/stuck name")
  public void stuckCommand(MessageReceiver caller, String[] args) {
    Player victim = Canary.getServer().getPlayer(args[1]);
    if (victim != null) {
      stuck(victim);
    }
  }
  public void stuck(Player player) {
    Location loc = player.getLocation();
    int playerX = (int) loc.getX();
    int playerY = (int) loc.getY();
    int playerZ = (int) loc.getZ();
    loc.setX(playerX + 0.5); loc.setY(playerY); loc.setZ(playerZ + 0.5);
    player.teleportTo(loc);

    int[][] offsets = {
     //x,  y,  z
      {0,  -1, 0},
      {0,  2,  0},
      {1,  0,  0},
      {1,  1,  0},
      {-1, 0,  0},
      {-1, 1,  0},
      {0,  0,  1},
      {0,  1,  1},
      {0,  0, -1},
```

```
      {0,  1, -1},
    };

    for(int i = 0; i < offsets.length; i++) {
      int x = offsets[i][0];
      int y = offsets[i][1];
      int z = offsets[i][2];
      setBlockAt(new Location(x+playerX, y+playerY, z+playerZ),
        BlockType.Stone);
    }
  }
}
```

Compila il plug-in `Stuck`, eseguilo e prova a usarlo. Cosa accade se il giocatore è in piedi sul terreno o in alto, per aria? Cosa succede dentro ai blocchi dal punto di vista del giocatore?

Prova da solo

Nel plug-in `Stuck`, abbiamo imprigionato un giocatore utilizzando il numero minimo di blocchi necessario a rinchiuderlo. Dall'esterno, però, la forma della gabbia appare un po' bizzarra.

Quello che dovremo fare è aggiungere altri blocchi in modo che il giocatore sia racchiuso in un rettangolo solido, largo e profondo tre blocchi e alto quattro. Aggiungi i blocchi supplementari all'elenco degli offset, quindi ricompila ed esegui per vedere se li hai collocati nei punti giusti. Dall'esterno, la "prigione" di pietra dovrebbe apparire come un unico blocco rettangolare.

Modificare le entità

Le entità sono abbastanza diverse dai blocchi. Tanto per cominciare, esistono diversi tipi di entità con tante abilità (per noi funzioni) differenti. Nel caso dei blocchi, l'unica cosa che puoi fare è cambiare il loro tipo e a volte aggiungere delle informazioni (come su un cartello). Le entità sono molto più complesse.

Tutte le entità hanno le capacità descritte in `net.canarymod.api.entity.Entity`. Ogni oggetto `Entity` include le seguenti funzioni utili.

- `getLocation()` restituisce la `Location` dell'entità.
- `setFireTicks(int ticks)` imposta il tempo durante il quale un'entità va a fuoco.
- `setRider(Entity rider)` imposta il *rider* dell'entità (cioè chi la cavalca).
- `spawn()` diffonde questo tipo di `Entity` nel mondo.
- `teleport(Location location)` teletrasporta l'entità in una nuova posizione.

A seconda del tipo di entità, hai a disposizione altre funzioni interessanti con cui giocare. Quelle che seguono sono specifiche delle entità viventi (`net.canarymod.api.entity.living.EntityLiving`).

- `getItemInHand()` restituisce l'item posseduto dall'entità.
- `setAttackTarget(LivingBase living-base)` imposta il bersaglio dell'attacco dell'entità.
- `getHealth()` restituisce un valore `double` dello stato di salute dell'entità. Può andare da zero (la morte) fino a un valore restituito da `getMaxHealth()`.
- `setHealth(double health)` imposta lo stato di salute. Zero è la morte, e non causa danni.
- `kill()` uccide l'entità provocando dei danni (perdita del bottino e altro).

Potresti aver notato che tutte queste funzioni sono dichiarate in `EntityLiving`. È qui dove Java diventa un po' complesso. Gli oggetti entità "familiari" incorporano molte ricette (classi) genitore diverse. Per esempio, una `Cow` (una mucca) è tanto un oggetto `Ageable` (che può avere un'età), quanto un `EntityAnimal` (un animale), un `EntityLiving` (un essere vivente) e ovviamente un'`Entity` (un'entità pura e semplice).

Questo significa che utilizza funzioni da tutti questi genitori. Per esempio,

poiché `Cow` eredita da `Ageable`, ottieni funzioni di cui puoi modificare una proprietà per cambiare l'età della mucca.

Un `Player`, invece, non usa `Ageable`, quindi non puoi trasformare giocatori in bambini, anche se si comportano come tali. Per contro, per un `Player` sono disponibili vari set di funzioni, comprese quelle per impostare e ottenere il livello di esperienza del giocatore, il livello di cibo, l'inventario e così via.

Generare le entità

Puoi usare numerose funzioni per generare più entità e creature, oltre che oggetti di gioco come un Ender Pearl. (In modalità Sopravvivenza, facendo clic con il tasto destro del mouse su un Ender Pearl vieni trasportato al suo punto d'impatto.) Per creare nuove entità nel mondo usiamo le funzioni definite nel nostro helper `EZPlugin` invece di scrivere direttamente il codice. Questo non perché il codice sia particolarmente complesso o difficile da capire; semplicemente le poche righe di codice sono le stesse, quindi ha senso ricorrere a una funzione helper. In questo modo ti basterà usare la chiamata alla funzione helper di una sola riga invece che utilizzare molte righe di codice duplicato. Vediamo da vicino cosa fa questa funzione.

Quello che segue è il modo che ho usato per generare mucche, calamari e altre entità del genere.

EZPlugin/src/com/pragprog/ahmine/ez/EZPlugin.java

```
public static EntityLiving spawnEntityLiving(Location loc, EntityType type) {
  EntityFactory factory = Canary.factory().getEntityFactory();
  EntityLiving thing = factory.newEntityLiving(type, loc);
  thing.spawn();
return thing;
}
```

È un po' caotico ma è lo schema comune di Java. L'idea è che prima si ottiene un oggetto *factory*, in questo caso `EntityFactory`. Un oggetto factory è un oggetto "di produzione", cioè che genera le cose per te, in questo caso nuovi oggetti `EntityLiving`. Ma, nel mondo di Minecraft, creare un nuovo oggetto (anche un oggetto `Cow`) non è sufficiente perché questo esista. Bisogna dire all'oggetto di diffondersi (`spawn`).

Diffondere delle particelle è un po' più semplice.

EZPlugin/src/com/pragprog/ahmine/ez/EZPlugin.java

```
public static void spawnParticle(Location loc, Particle.Type type) {
  loc.getWorld().spawnParticle(new Particle(loc.getX(),
    loc.getY(), loc.getZ(), type));
}
```

Qui ci basta usare la funzione `spawnParticle` in `World` e passare una nuova `Particle` (particella) inizializzata con le singole coordinate x, y e z. Anche in questo caso il codice non è complesso, ma una funzione helper ci aiuta a mantenere il codice più

semplice e chiaro da leggere.

Anche la funzione che usiamo per impostare i tipi di blocchi è immediata.

EZPlugin/src/com/pragprog/ahmine/ez/EZPlugin.java

```
public static void setBlockAt(Location loc, BlockType type) {
  loc.getWorld().setBlockAt(loc, type);
}
```

Sebbene un `Block` abbia il proprio `setType`, questo non è sufficiente per ottenere la modifica che vorremmo portare nel mondo; in alternativa, usiamo la funzione `setBlockAt()` di `World`. Ricorrendo a una funzione helper, saremo sicuri di impiegare sempre la versione corretta.

Plug-in: FlyingCreeper

Quello che vedremo ora è un plug-in che genera due entità: un pipistrello e un Creeper. Faremo in modo che il Creeper cavalchi il pipistrello e poi lo renda invisibile usando un effetto provocato da una pozione. Il risultato sarà un terribile Creeper volante.

Studiamo il cuore del plug-in. Nota che questa volta non uso le funzioni helper, bensì direttamente le chiamate factory, poiché qui impiego una versione di `spawn` leggermente diversa.

FlyingCreeper/src/flyingcreeper/FlyingCreeper.java

```
Location loc = me.getLocation();
loc.setY(loc.getY() + 5);

EntityFactory factory = Canary.factory().getEntityFactory();
EntityLiving bat = factory.newEntityLiving(EntityType.BAT,  loc);
EntityLiving creep = factory.newEntityLiving(EntityType.CREEPER, loc);
bat.spawn(creep);

PotionFactory potfact = Canary.factory().getPotionFactory(); PotionEffect potion =
  potfact.newPotionEffect(PotionEffectType.INVISIBILITY,
          Integer.MAX_VALUE, 1);
bat.addPotionEffect(potion);
```

Tutti gli oggetti `Entity` possono avere dei rider; in teoria, potresti anche cavalcare della TNT innescata (ma non te lo consiglio). Qui faremo in modo che il Creeper cavalchi il pipistrello generando il pipistrello con un Creeper che lo cavalca (l'argomento di `spawn`).

Successivamente dovremo rendere il pipistrello invisibile per rendere più convincente il Creeper volante. Fortunatamente, tutti gli oggetti `LivingEntity` (cioè le entità viventi) usano pozioni con effetti (gli effetti sono tutti elencati all'indirizzo http://bit.ly/1vdHPV2). Creiamo un nuovo effetto di pozione che ci permetterà di specificare il tipo di effetto, la sua durata e la potenza:

```
PotionEffect (PotionEffectType type, int duration, int amplifier)
```

Qui il tipo è `PotionEffectType.INVISIBILITY` e vogliamo che valga per sempre, per cui allunghiamo il più possibile il valore della sua durata: `Integer.MAX_VALUE`. Non esiste intero più grande. In questo caso la potenza non ci interessa particolarmente, perché non si può essere "più invisibili", quindi usiamo 1.

Infine, aggiungiamo la nuova pozione al pipistrello, che diventerà invisibile.

Congratulazioni! Ora sei il padrone orgoglioso di un Creeper volante. Buona

fortuna, e vola basso.

Se vuoi fare qualche passo in più, puoi tornare al Capitolo 6 e a `SquidBomb` per generare alcuni Creeper invisibili invece dei calamari. Potrebbe essere divertente.

Vedremo altri esempi di creazione e modifica delle entità nel prossimo paragrafo, dopo aver imparato ad ascoltare gli eventi del gioco.

Ascoltare gli eventi

Adesso viene il meglio. Sai come scrivere il codice per i comandi che vengono digitati dall'utente e come compiere azioni che influiscono sul mondo. È il momento di vedere come monitorare quanto accade nel mondo per rispondere al gioco automaticamente, senza digitare comandi o altro.

L'obiettivo è quello di impostare il codice in modo che le funzioni vengano chiamate quando nel gioco succede qualcosa di interessante. Nella funzione, puoi permettere che gli eventi si verifichino oppure puoi interromperli.

Ecco i quattro passi che dobbiamo compiere nel nostro plug-in di base per incorporare un *listener* (cioè un "ascoltatore").

1. Importare il listener del plug-in e le classi *hook* ("gancio") degli eventi.
2. Dichiarare il plug-in `implements PluginListener`.
3. Abilitare il plug-in perché ascolti gli eventi con `registerListener`.
4. Aggiungere il tag magico `@HookHandler`, contrassegnando la funzione come un *handler* (cioè un gestore) di eventi. L'evento che ti interessa va fornito come argomento.

Questo breve esempio mostra i quattro passaggi della procedura:

```
  import net.canarymod.hook.HookHandler;
1 import net.canarymod.plugin.PluginListener;
  // Importa qui la classe hook specifica

  // Aggiunge qui "implements PluginListener":
2 public class HelloWorld extends EZPlugin implements PluginListener {

    @Override
3   public boolean enable() { Canary.hooks().registerListener(this,  this);
    return super.enable(); // Chiama anche la versione genitore della classe.
    }

    // Ecco un listener di eventi:
4   @HookHandler
    public void anyname(SomeHook hookevent) {
           // Qui va del codice
    }
  }
```

La prima cosa da fare è importare la classe per l'evento che ci interessa (aggiungila da qualche parte al punto **1**).

Gli eventi disponibili sono centinaia, tutti elencati nella documentazione di Canary in `net.canarymod.hook` e nei suoi sottomenu. Immagina di voler fare qualcosa ogni volta che qualcuno nel gioco viene teletrasportato. Ti servirà la classe

`TeleportHook` nel package `net.canarymod.hook.player`, quindi devi iniziare con `import net.canarymod.hook.player.TeleportHook`.

La dichiarazione per il plug-in deve poi aggiungere le parole magiche `implements PluginListener`, come mostrato al punto **2**.

A seguire devi aggiungere la tua funzione `enable()` (cioè "abilita"), che chiama la versione genitore della classe, come mostrato a partire dal punto **3**. È una sorta di segmento di codice standard che dice semplicemente "permetti a questo plug-in di ascoltare gli eventi". Questo frammento di codice ti servirà una sola volta nel file, e funzionerà per tutti gli eventi che userai. Aggiungilo e proseguiamo.

Arriviamo al listener vero e proprio (punto **4**). `@HookHandler` è un'annotazione che dice a Java che la funzione successiva è speciale, proprio come abbiamo visto con `@Command`. Ricorda di anteporre sempre questa annotazione a un handler di eventi.

La funzione per il listener può avere qualsiasi nome (qui è `anyname`), mentre la lista degli argomenti deve essere più precisa: il tipo di eventi che elenchi in questo punto stabilisce dove la funzione verrà chiamata o addirittura se verrà chiamata.

Un handler di eventi per `TeleportHook` ha questo aspetto:

```
@HookHandler
public void myTeleportListener(TeleportHook event) {
    // Qui va del codice
}
```

La parte che conta non è `myTeleportListener`, ma `TeleportHook`. Come riporta la documentazione, questo oggetto hook di evento fornisce numerose funzioni interessanti.

- `getCurrentLocation()` restituisce la posizione da cui il giocatore viene teletrasportato.
- `getDestination()` ottiene la posizione in cui il giocatore viene teletrasportato.
- `getPlayer()` ottiene il giocatore.
- `setCanceled()` consente di annullare l'evento.

Per esempio, per impedire che niente venga teletrasportato da nessuna parte, dovremo scrivere

```
@HookHandler
public void myTeleportListener(TeleportHook event) {
    event.setCanceled();
```

```
}
```

Plug-in: BackCmd

Utilizziamo alcune di queste funzioni in un plug-in completo. Sotto trovi il percorso per il codice sorgente di `BackCmd`, che fornisce un unico comando `back` (cioè "indietro"). Trovalo e installalo:

```
$ cd Desktop
$ cd code/BackCmd
$ ./build.sh
```

Riavvia il server. Ora effettua un teletrasporto verso un paio di posizioni diverse, usando il comando `/tp` in modalità Creativa oppure facendo clic con il tasto destro del mouse su un Ender Pearl in modalità Sopravvivenza.

Digita `/back` e verrai teletrasportato indietro alla tua ultima posizione.

Questo plug-in ascolterà gli eventi per seguire dove ti trovi e ti permetterà di tornare alle tue ultime posizioni, in ordine.

BackCmd/src/backcmd/BackCmd.java

```
 package backcmd;

 import java.util.ArrayList;
 import java.util.HashMap;
 import java.util.List;
 import java.util.Stack;
 import net.canarymod.plugin.Plugin;
 import net.canarymod.logger.Logman;
 import net.canarymod.Canary;
 import net.canarymod.commandsys.*;
 import net.canarymod.chat.MessageReceiver;
 import net.canarymod.api.entity.living.humanoid.Player;
 import net.canarymod.api.world.position.Location;
 import net.canarymod.hook.HookHandler;
 import net.canarymod.hook.player.TeleportHook;
 import net.canarymod.plugin.PluginListener;
 import com.pragprog.ahmine.ez.EZPlugin;

1 public class BackCmd extends EZPlugin implements PluginListener {

2   private static List<Player> isTeleporting = new ArrayList<Player>();
    private static HashMap<String, Stack<Location>> playerTeleports =
        new HashMap<String, Stack<Location>>();

    @Override
3   public boolean enable() {
      Canary.hooks().registerListener(this,  this);
      return super.enable(); // Chiama anche la versione genitore della classe.
    }

4   public boolean equalsIsh(Location loc1, Location loc2) {
      return ((int) loc1.getX()) == ((int) loc2.getX()) &&
             ((int) loc1.getZ()) == ((int) loc2.getZ());
    }

5   @HookHandler
```

```
      public void onTeleport(TeleportHook event) {
      Player player = event.getPlayer();
      if (isTeleporting.contains(player)) {
        isTeleporting.remove(player);
      } else {
        Stack<Location> locs = playerTeleports.get(player.getName());
        if (locs == null) {
          locs = new Stack<Location>();
        }
        locs.push(player.getLocation());
        locs.push(event.getDestination());
        playerTeleports.put(player.getName(),  locs);
      }
    }

    @Command(aliases = { "back" },
              description = "Go back to previous places that you teleported to.",
              permissions = { "" },
              toolTip = "/back")
    public void backCommand(MessageReceiver caller, String[] args) {
      if (caller instanceof Player) {
        Player me = (Player)caller;

        Stack<Location> locs = playerTeleports.get(me.getName());

       if (locs != null && !locs.empty()) {
         Location loc = locs.peek();
         while (equalsIsh(loc, me.getLocation()) && locs.size() > 1) {
           locs.pop();
           loc = locs.peek();
         }
         isTeleporting.add(me);
         me.teleportTo(loc);
       } else {
         me.chat("You have not teleported yet.");
       }
     }
   }
 }
```

Poiché questo plug-in ascolta gli eventi, dovrai eseguire alcune azioni specifiche.

- Al punto **1**, dichiara che questo plug-in implementa un listener (`implements PluginListener`).
- Al punto **3**, abilita il plug-in per gli eventi.
- Al punto **5**, aggiungi l’annotazione `@HookHandler` e dichiara che questo listener ascolterà gli eventi `TeleportHook`.

Ogni volta che un giocatore viene teletrasportato, la funzione `onTeleport` verrà chiamata. La prima cosa da fare è ottenere l’oggetto `Player` corretto, che è memorizzato nell’`event` che ci è stato passato; così sappiamo con cosa abbiamo a che fare.

A seguire, cerchiamo nella lista `isTeleporting` se questo giocatore è qualcuno che

stiamo teletrasportando.

Se dai un'occhiata all'inizio del plug-in, attorno al punto **2**, vedrai due variabili dichiarate come `private static`. Significa che nessun altro può vederle, e i valori resteranno disponibili finché questo comando è in esecuzione. In questo caso `isTeleporting` e `playerTeleports` tengono traccia dei giocatori e delle posizioni da cui e verso cui vengono teletrasportati.

Controlliamo se la lista `isTeleporting` contiene già il giocatore. Se non lo contiene, aggiungiamolo (il giocatore viene teletrasportato). Se invece la lista contiene già il giocatore, ignoreremo l'evento (e rimuoveremo il giocatore dalla lista). Questo vuol dire che abbiamo generato noi l'evento, e non vogliamo che il nostro teletrasporto attivi quello generale, perché altrimenti continuerebbe a farlo all'infinito. Per questo dobbiamo cancellare l'evento di teletrasporto.

Se il giocatore è già stato teletrasportato in precedenza, otterremo alcune `locs` (un gruppo di posizioni) dall'hash `playerTeleports` usando il giocatore come chiave. Aggiungeremo poi questa posizione alla lista esistente. Se questa è la prima volta che il giocatore viene teletrasportato, creeremo una nuova lista di `Location` e aggiungeremo questo luogo come prima voce.

Questa, però, non è una vera e propria lista, ma uno `Stack`. Uno `Stack` funziona come una pila di libri: aggiungi un libro in cima alla pila e quando ne togli un altro, il primo libro scende di una posizione. Questo è il modello che vogliamo applicare alla nostra lista di luoghi di teletrasporto. Durante il teletrasporto, ogni posizione viene aggiunta in cima alla pila; da lì si scende verso quella successiva, poi verso quella dopo ancora e così via.

Per aggiungere un oggetto in cima alla pila, lo si spinge (`push`) dentro. Per ottenere il valore e rimuovere l'oggetto dalla cima, lo si estrae (`pop`). Per controllare semplicemente il valore dell'oggetto più in alto senza rimuoverlo, gli si dà una sbirciata (`peek`).

Mentre i giocatori vengono teletrasportati, teniamo traccia delle loro posizioni memorizzandole in uno stack nella variabile `playerTeleports`. Possiamo accedere allo stack utilizzando l'oggetto `Player` come chiave. La Figura 9.2 mostra uno schema di questo meccanismo.

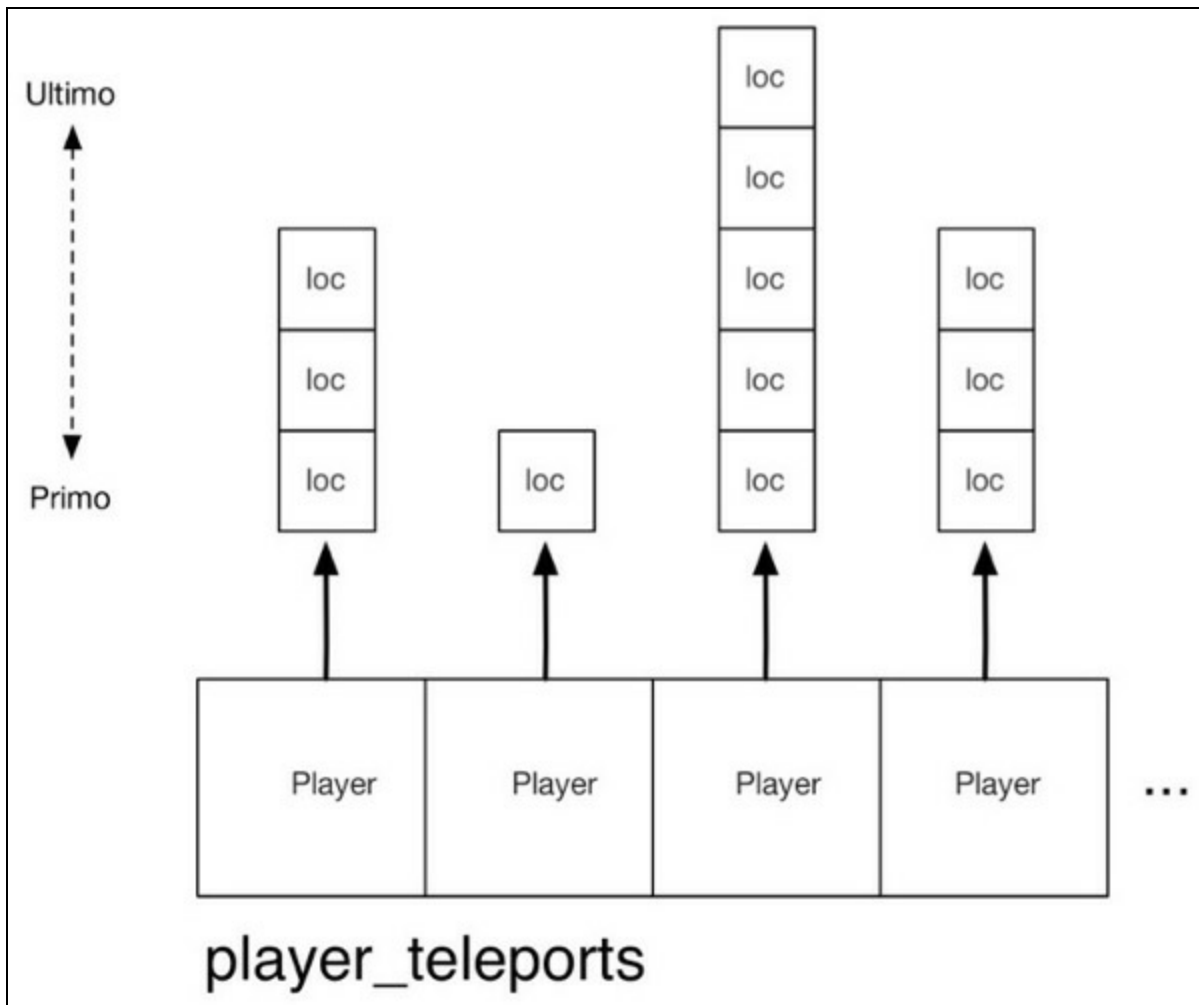



Figura 9.2 playerTeleports è un hash con le chiavi dei giocatori e i valori dello stack.

Per finire, torniamo a `backCommand`. Quando eseguiamo un comando `back`, otteniamo una lista delle posizioni di teletrasporto del giocatore (se ce n'è uno), e sbirciamo la posizione in cima allo stack, salvandola momentaneamente in `loc`.

Probabilmente abbiamo effettuato un teletrasporto in questo punto e non ci siamo ancora spostati. Nel nostro caso vogliamo però tornare al punto precedente, non restare qui. Facciamo un test, allora: se siamo ancora nella stessa posizione dell'ultimo teletrasporto – o lì vicino –, rimuoviamo quella posizione dalla lista delle `locs` usando `pop` e impostiamo `loc` alla posizione successiva nello stack.

In entrambi i casi otteniamo una `loc` che punta alla posizione verso cui vogliamo eseguire il teletrasporto. Aggiungiamo il giocatore alla lista `isTeleporting` e teletrasportiamolo nella nuova posizione.

Ma cosa intendiamo per "o lì vicino"? Osserva che al punto **4** abbiamo

dichiarato una funzione helper chiamata `equalsIsh`. Questa funzione controlla due posizioni per vedere se le coordinate x e z sono entrambe all'interno dello stesso blocco. Convertendo i valori a virgola mobile in interi tramite la parola chiave `(int)`, eliminiamo la parte decimale della coordinata. Questo ci interessa solo se ci troviamo nello stesso blocco, e non in una posizione leggermente diversa dentro allo stesso blocco.

Prova da solo

In questo esercizio creerai il plug-in nuovo di zecca `FireBow` che usa un listener.

Nel momento in cui colpirà il bersaglio, la tua freccia incendiaria provocherà una vasta esplosione che aprirà un cratere. Per farlo, ascolta l'evento `ProjectileHitHook` e chiama `world.makeExplosion`. Per creare un cratere più largo, usa un valore grande per la potenza, come `100.0f`. Non dimenticare di rimuovere l'evento freccia; in questo modo non ci sarà una freccia, ma solo un'esplosione.

Un suggerimento: puoi ottenere la freccia dall'evento `ProjectileHitHook` usando `event.getProjectile()`.

Trovi il codice completo del mio esempio in *code/FireBow*. Prima prova da solo; se rimani bloccato, guarda la mia soluzione. Ho aggiunto anche due comandi che ti consentono di abilitare e disabilitare il comportamento dell'arco.

Controllare i permessi

Quando scrivi un plug-in, potresti voler limitare l'uso dei comandi che hai creato: potresti farli eseguire solo durante certe operazioni, o magari hai a che fare con diversi tipi di giocatori o squadre e vuoi che ciascuno possa eseguire solo determinati comandi e non altri.

Per esempio, immagina di voler limitare l'uso dell'arco solo a un gruppo selezionato di giocatori; chiamiamo questa indicazione `commands.firebow.enable`.

Puoi usare il campo `permissions` nell'annotazione `@Command` per verificare se il giocatore che sta eseguendo il comando è uno di quelli selezionati:

```
@Command(aliases = { "nofirebow" },
          description = "Disable firebow behavior",
          permissions = { "commands.firebow.enable" },
          toolTip = "/nofirebow")
```

Ora solo i giocatori che hanno il permesso per `commands.firebow.enable` possono lanciare le frecce infuocate. Eccezionale. Ma come si fa ad assegnare questi permessi?

Impostare e gestire i permessi

L'impostazione e la gestione dei permessi possono essere impegnative se lavori su un grosso server con molti utenti. Per aiutarti, puoi definire dei gruppi o delle classi di utenti usando i comandi dei permessi di Canary.

Canary fornisce una serie comandi direttamente nel gioco, tra cui alcuni per impostare i permessi per giocatori e gruppi e per controllare di quali permessi dispone qualcuno. Puoi eseguire questi comandi nella console del server o dal client. Ecco alcuni esempi.

- `/playermod permission add` *nomeGiocatore nomePermesso*: concede il *nomePermesso* a *nomeGiocatore*.
- `/playermod permission check` *nomeGiocatore nomePermesso*: verifica se *nomeGiocatore* ha il permesso *nomePermesso*.
- `/playermod group set` *nomeGiocatore nomeGruppo*: aggiunge *nomeGiocatore* al gruppo *nomeGruppo*.
- `/groupmod permission list` *nomeGruppo*: elenca i permessi per *nomeGruppo*.

- `/groupmod permission add` *nomeGruppo nomePermesso*: aggiunge il permesso *nomePermesso* ai membri di *nomeGruppo*.
- `/groupmod permission check` *nomeGruppo nomePermesso*: verifica se i membri di *nomeGruppo* hanno il permesso *nomePermesso*.

Per esempio, nel nostro plug-in, per dare a `jack37` i permessi per usare i comandi di `/firebow`, dobbiamo digitare i seguenti comandi nella console:

```
playermod permission add jack37 commands.firebow.enable
```

Consulta la documentazione di Canary per altri dettagli.

Per continuare

In questo capitolo hai acquisito molte capacità: ora sei in grado di modificare i blocchi di Minecraft e generare entità, sai gestire i permessi di un plug-in, e soprattutto sai come ascoltare gli eventi interessanti che accadono nel gioco e rispondere a essi. Questo è il cuore dei plug-in, quello che può cambiare il modo in cui funziona il mondo di Minecraft, agendo automaticamente sugli eventi man mano che si verificano.

Nel prossimo capitolo vedremo come programmare gli eventi nel futuro, e realizzeremo uno spara mucche. Ci vediamo là.

La tua toolbox

60%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.
- Usare gli oggetti Java.
- Usare le istruzioni `import` per i package Java.
- Usare `new` per creare gli oggetti.
- Aggiungere un nuovo comando a un plug-in.
- Lavorare con gli oggetti `Location`.
- Trovare blocchi ed entità.
- Usare le variabili locali.
- Usare le variabili globali a livello di classe.
- Usare gli `ArrayList`.
- Usare le `HashMap`.
- Usare `private` e `public` per controllare la visibilità.
- Modificare i blocchi di Minecraft.
- Modificare e generare le entità.
- Monitorare gli eventi di gioco e rispondere.
- Gestire i permessi dei plug-in.

Programmare le attività

In questo capitolo parleremo di come far succedere in Minecraft cose che non sono una risposta immediata a un evento o a un comando eseguito dall'utente; vedremo cioè come programmare alcune attività che devono avvenire nel futuro (anche continuativamente) in modo autonomo. Questo meccanismo ti aiuterà a implementare azioni che sembrano accadere di per sé, come un attacco ai Creeper o ad altri nemici.

Per quanto riguarda Java, vedremo come creare classi personali in file separati, e studieremo alcuni dei problemi che si trova ad affrontare Java quando svolge più compiti contemporaneamente.

Cosa accade e quando?

Quando parliamo di far eseguire qualcosa “in seguito”, dobbiamo affrontare una scomoda verità: il mondo non smette di cambiare solo perché il nostro codice sta girando. I computer possono fare più cose nello stesso tempo, e lo fanno in continuazione. Purtroppo, però, la maggior parte del codice non è scritta con lo stesso criterio.

Pensa ad alcuni dei plug-in che hai visto finora. Cosa succederebbe se, nel bel mezzo dell’esecuzione del codice, un altro giocatore digitasse lo stesso comando e il codice ricominciasse a essere eseguito dall’inizio?

Osserva la Figura 10.1. Ogni freccia e segmento di codice rappresenta un insieme di istruzioni svolte simultaneamente dal computer. Ognuno di questi insiemi è chiamato *thread*, cioè filo, come il filo di un discorso.

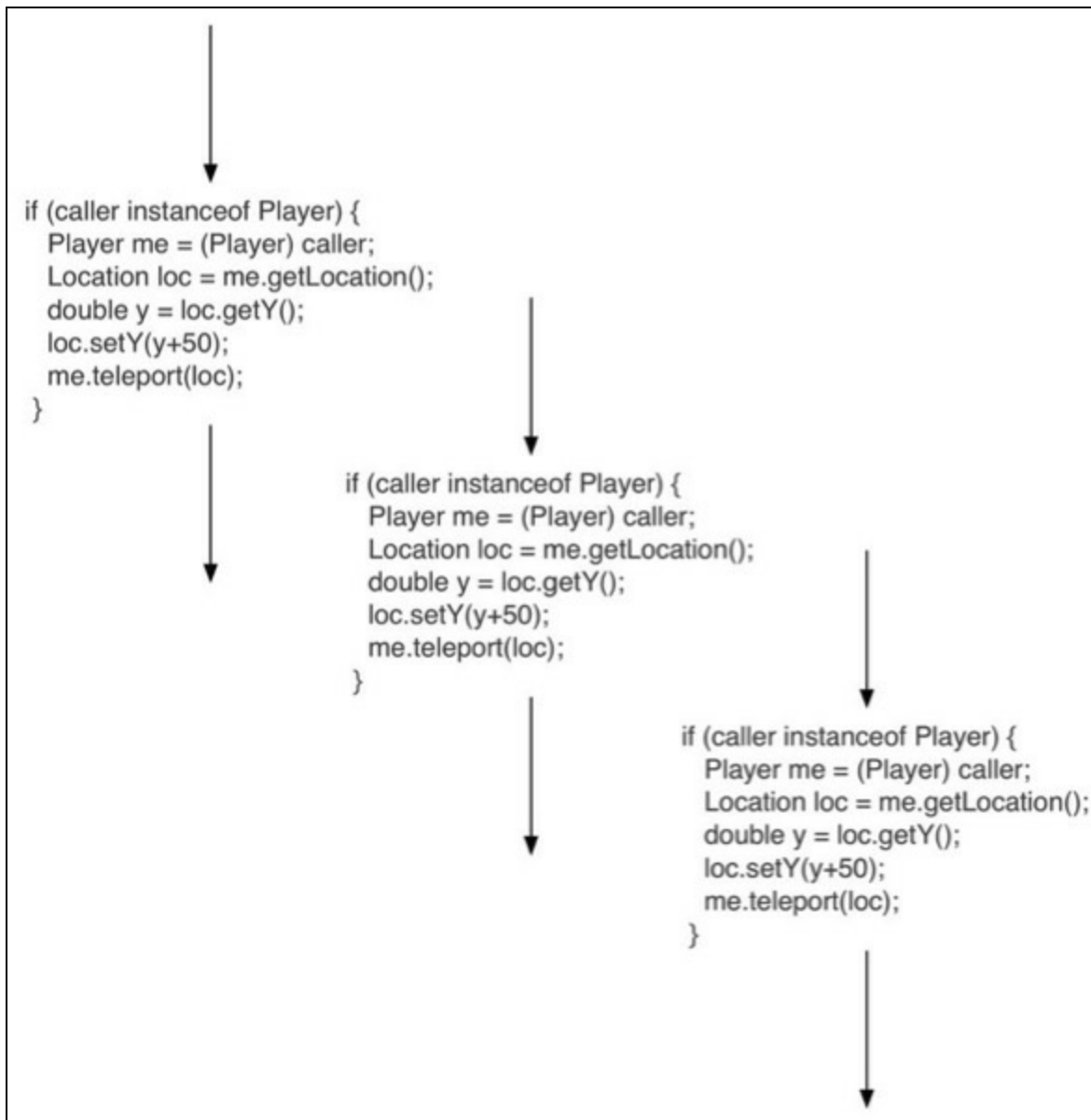



Figura 10.1 Più thread eseguono il codice.

Pensiamo a un computer che esegue il codice una volta, come nella parte sinistra della figura, svolgendo le istruzioni una dopo l'altra, dall'alto verso il basso, e senza che avvenga altro nello stesso tempo. In questo modo si avrebbe un thread di esecuzione (come una conversazione).

Nella realtà, però, il computer può eseguire una serie di diversi thread contemporaneamente. Il nostro codice è in grado di lavorare così? Cosa succederebbe se venisse eseguito da più thread?

Se usi solo variabili locali che non cercano di modificare nulla di comune nel mondo di Minecraft, probabilmente non ci saranno problemi: ciascuna versione

del codice in esecuzione avrà la sua copia di variabili, senza far danni. Se però usi una variabile statica, qualcosa come una grande `HashMap` per tenere traccia di giocatori e valori, allora avrai due frammenti di codice che leggono e scrivono dal codice nello stesso momento, e questo in genere porta al disastro.

Immagina di leggere il livello di salute di un giocatore come 50, ma che in un altro thread sia impostato a 0, uccidendolo. Non sapendolo, sottrai 10 dalla salute del giocatore e la reimposti a 40. L'altro thread pensa di aver ucciso il giocatore ma, invece di 0, scopre che la sua salute è 40! Questo è un esempio semplice e innocuo, ma possono verificarsi situazioni peggiori che generano bachi o provocano il fallimento del codice (e del server).

Esistono diversi modi per scrivere un codice che possa eseguire più thread in tutta sicurezza; tuttavia, molte librerie, e la maggior parte dell'API Canary, non sono concepite così.

Vale la pena ripeterlo: *l'API Canary e il nostro codice di plug-in che la utilizza non sono studiati per eseguire più thread.* Il codice si interromperà.

Se vogliamo eseguire un frammento di codice "in seguito", non possiamo permettere che ciò avvenga casualmente nel tempo. In particolare, non possiamo permettere che un segmento del nostro codice venga eseguito nello stesso momento di un altro, o nello stesso momento del codice del server.

Dovremo invece consentire al server di determinare quando eseguire il codice. In questo modo potrà farlo come se un giocatore digitasse un comando o come se si rispondesse a un evento del gioco; quella sarà l'unica cosa in esecuzione in quell'istante. Un'attività di questo genere è chiamata *sincrona*, ed è quanto imparerai a impostare in questo capitolo.

Per farlo, però, dobbiamo prima capire come creare delle classi personali.

Inserire il codice in una classe indipendente

Per iniziare, cerchiamo di essere più precisi nella denominazione delle cose. Nel codice sorgente Java, un plug-in viene dichiarato come una `public class`, che è il modo di Java per dire “questa è una ricetta. Posso creare oggetti di questa classe durante l’esecuzione”. Fino a questo momento abbiamo parlato approssimativamente di “ricette” e “ricette genitore”; qui inizieremo a chiamarle con il loro nome Java, cioè *classi*. Le ricette sono classi.

Finora abbiamo un po’ imbrogliato: abbiamo messo tutto il nuovo codice in una classe del plug-in, in un file di codice sorgente. Quando il server di Minecraft è in esecuzione, crea un oggetto dal file della classe e lo usa.

Se però osservi altri plug-in in Internet, per esempio, noterai che spesso un plug-in (e altri programmi più grandi) è costituito da più classi diverse che collaborano. Il file principale della classe contiene il plug-in stesso, mentre gli altri file della classe possono contenere oggetti correlati, attività da programmare o un altro tipo di codice helper di cui il plug-in potrebbe aver bisogno. Nei nostri esempi abbiamo aggiunto il codice direttamente nella classe principale, ma nel caso delle attività da programmare lo definiremo nelle classi specifiche.

Creare un nuovo file sorgente è semplice: basta che nel tuo editor di testi selezioni *File* > *Salva* (o un comando simile). Una volta che l’hai salvato nella cartella corrente, il sistema vedrà il nuovo file e saprà cosa farci. Ma non è sufficiente. Devi imparare a dire al plug-in che esiste il nuovo file, e far sapere al nuovo file che c’è il plug-in.

Cosa inserire in una classe?

Ai programmatori principianti spesso viene spiegato che una classe è come una scatola che contiene funzioni e dati. L’idea non è delle migliori: potresti finire con l’avere una soffitta o un garage disordinato, pieno di scatole messe una sopra l’altra che straripano di oggetti.

Il metodo ideale è un altro. Ogni classe che crei dovrebbe essere responsabile di un’unica cosa; in altre parole, dovrebbe avere un solo motivo per cambiare. Se

è responsabile di più cose, ha anche più ragioni per cambiare: è fragile, e il codice fragile porta con sé solo guai.

L'API Canary ci guida nella giusta direzione. Possiamo creare la nostra classe del plug-in, ma per poter impostare un'attività programmata ci servirà una seconda classe fatta apposta, così che alla fine avremo due classi: il plug-in e l'attività. E dovremo farle collaborare.

Vediamo cosa occorre per creare un'attività programmabile perché venga eseguita in futuro (quando il server ritiene che ciò possa avvenire). Prima analizzeremo i singoli frammenti, poi assembleremo il tutto in un plug-in.

Creare un’attività eseguibile

Quello che segue è il codice più semplice per un’attività che invia un messaggio all’utente. Non fa granché, ma puoi usarlo come modello di partenza per creare altre attività personali.

Aggiungi il codice che desideri nel corpo della funzione `run`, modifica il nome del package e inserisci quante funzioni di importazione ti occorrono. (Ricorda che le più comuni sono elencate nell’Appendice F, mentre altre le trovi nella documentazione di Canary o di Java.)

```
1 package examplepackage;
  import net.canarymod.Canary;
  import net.canarymod.tasks.ServerTask;
  import com.pragprog.ahmine.ez.EZPlugin;
2 public class ExampleTask extends ServerTask {

  private final EZPlugin plugin;

3 public ExampleTask(EZPlugin plugin) {
    super(Canary.getServer(), 0, true); // Ritardo, isContinuous
    this.plugin = plugin;
    // Puoi mantenere un riferimento al plug-in come ho fatto qui
    // oppure a qualsiasi altra variabile che passi
  }

  public void run() {
    // Fa qualcosa di interessante...
    Canary.instance().getServer().broadcastMessage(“Surprise!”);
    }
  }
```

In questo caso inserisci il codice in un suo file, che deve chiamarsi `ExampleTask.java` (per coincidere con il nome nella riga al punto **2**) ed essere in una cartella nominata come il package (per coincidere con il nome al punto **1**).

Fin qui tutto bene. Ma cosa succede con questa funzione che non ha un tipo di ritorno e che si chiama `ExampleTask` (**3**), cioè esattamente come la nostra classe? Come ricorderai, questo genere di funzione è detto *costruttore*, ed è quella che Java chiama quando crea una nuova `ExampleTask`.

Per creare la nostra `ExampleTask`, dobbiamo passare un `EZPlugin` a `new` (che viene passata al costruttore). In altre parole, dobbiamo assegnarle l’oggetto principale del tuo plug-in. In questo modo l’attività potrà recuperare tutti gli elementi del server di cui potrebbe aver bisogno, come il mondo, il gioco e così via. Non è obbligatorio, ma è utile.

Ora che hai un’attività che fa qualcosa di interessante (più o meno), come fare a

programmarla?

Programmare per l’esecuzione futura

Torniamo al plug-in principale: una volta definita l’attività nella sua classe, puoi creare un oggetto di quella classe dal plug-in usando `new` e programmarla in modo che venga eseguita nel futuro utilizzando `addSynchronousTask()`:

```
ExampleTask task = new ExampleTask(plugin);
Canary.getServer().addSynchronousTask(task);
```

`plugin` dovrebbe essere una variabile impostata nel plug-in principale. Se inserisci questo codice in una funzione non statica del plug-in, puoi usare la parola chiave `this` di Java, che significa “questo oggetto del plug-in”.

Se invece inserisci questo codice in una funzione statica, ti servirà una variabile statica (come `plugin`) che è stata impostata in precedenza nel plug-in. Per esempio, in una funzione di oggetto (un *metodo*) come il tuo costruttore o come `enable`, dovresti impostare `plugin = this`.

Volendo puoi limitarti alla variabile `task`, ma non è indispensabile. Con o senza di essa, hai creato un’`ExampleTask` che programmerà la funzione `run` in modo che venga eseguita in qualche momento nel futuro.

Una sola esecuzione o un’esecuzione continua

Puoi specificare come dovrà essere eseguita l’attività impostando un ritardo finché non parte e stabilendo se deve essere eseguita una sola volta o in continuazione. Dai un’occhiata alla prima cosa che avviene nel costruttore: la chiamata a `super` al punto **3** chiamerà a sua volta il costruttore nella classe genitore `ServerTask`, con tutte le informazioni necessarie.

Il costruttore `ServerTask` prende tre argomenti: il server di per sé, un ritardo di attesa prima dell’esecuzione e un flag `boolean` che dovrà essere `true` se l’attività verrà eseguita in continuazione e `false` se dovrà essere eseguita una volta sola. Nel primo caso, l’attività proseguirà finché non l’annulli.

Possiamo modificare l’esempio precedente stabilendo un ritardo di 60 secondi prima dell’esecuzione singola:

```
super(Canary.getServer(), 1200, false);
```

A 20 tick al secondo, 1200 tick corrisponderanno a circa 60 secondi (un minuto).

Plug-in: CowShooter

Adesso disponiamo di elementi sufficienti per creare un plug-in davvero divertente: il `CowShooter` (lo spara mucche). Se nel gioco sei in possesso di un pezzo di cuoio, con un clic puoi sparare una mucca in fiamme nel mondo. Quando la mucca colpisce il terreno, esplode in una palla di fuoco e rilascia bistecche.

Questo plug-in è leggermente diverso da quelli che abbiamo visto finora; non c'è una sezione `@Command`, ed è guidato esclusivamente dagli eventi.

Un problema con le mucche in fiamme è che non rimangono in fiamme per sempre né a lungo. In genere una mucca prende fuoco, muore, e il fuoco si estingue. A noi invece serve una mucca che resti in fiamme più a lungo mentre vola in aria.

Per farlo succedere, programmiamo un'attività. L'attività continuerà a ripetersi finché la mucca rimane in aria, e nel frattempo potremo mantenere la mucca in vita e in fiamme fino al momento in cui colpisce il terreno.

Ecco il codice per il listener di eventi principale.

CowShooter/src/cowshooter/CowShooter.java

```
 @HookHandler
 public void onInteract(ItemUseHook event) {

   Player player = event.getPlayer();

   if (player.getItemHeld().getType() == ItemType.Leather) {
     Location loc = player.getLocation();
     loc.setY(loc.getY() + 2);

1    Cow victim = (Cow)spawnEntityLiving(loc, EntityType.COW);

     Canary.getServer().addSynchronousTask(new CowTask(victim));

     fling(player, victim, 3);
     victim.setFireTicks(600);
   }
 }
```

L'evento `ItemUseHook` può indicare che il giocatore ha usato uno di più elementi, quindi dovremo controllare e vedere se ha il cuoio. Se la risposta è no, ignoriamo l'evento e continuiamo come se niente fosse.

Se invece la risposta è sì, useremo il metodo helper `fling` per aumentare la velocità della mucca: vogliamo farla volare nella direzione in cui il giocatore sta guardando. Alziamo la velocità moltiplicandola per 3, che potrebbe essere una buona misura (a volte bisogna sperimentare). Partendo dal punto **1**, generiamo una

mucca, aggiungiamo la nuova attività programmata, impostiamo la velocità della mucca usando `fling` e diamole fuoco.

Come già detto, il problema con le mucche (e con qualsiasi oggetto) in fiamme è che non rimangono a lungo in questo stato, ma bruciano e muoiono. Ecco di cosa si occuperà la nostra attività: mantenerle in vita finché non toccano terra.

Questa è la classe separata che contiene l'attività eseguibile:

CowShooter/src/cowshooter/CowTask.java

```
package cowshooter;

import net.canarymod.Canary;
import net.canarymod.api.entity.EntityType;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.entity.living.animal.Cow;
import net.canarymod.api.world.position.Vector3D;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.tasks.ServerTask;
import com.pragprog.ahmine.ez.EZPlugin;

public class CowTask extends ServerTask {

    private Cow cow;

    public CowTask(Cow myCow) {
        super(Canary.getServer(), 0, true); // ritardo, isContinuous
        cow = myCow;
    }

    public void run() {
      if (cow.isOnGround()) {
        Location loc = cow.getLocation();
        cow.setHealth(0);
        cow.kill(); cow.getWorld().makeExplosion(cow,
          loc.getX(), loc.getY(), loc.getZ(),
          2.0f, true);
        Canary.getServer().removeSynchronousTask(this);
      } else {
        cow.setFireTicks(600);
        cow.setHealth((float)cow.getMaxHealth());
      }
    }
  }
```

COME FACCIO A PROCURARMI UN PEZZO DI CUOIO?

Se non hai un pezzo di cuoio a portata di mano, ecco come puoi procurartelo.

- Entra in modalità Creativa digitando `/gamemode c`.
- Premi il tasto E.
- L'icona nell'angolo superiore destro è una casella di ricerca. Facci clic sopra.
- Inizia a digitare `leather`; dovresti veder apparire il materiale Leather (cuoio, appunto).
- Fai clic una volta sul materiale e poi una volta sulla barra in basso per inserirlo nell'inventario.
- Chiudi l'inventario premendo il tasto Esc.

- Premi il numero corrispondente alla casella in cui hai inserito il cuoio (da 1 a 9) per selezionarlo.

Quando cerchi di entrare in modalità Creativa potresti vedere un errore che dice che non hai il permesso di farlo. Per risolvere il problema, devi assegnarti i privilegi di operatore (*op*).

Se la mucca ha colpito il terreno (puoi verificarlo con `cow.isOnGround()`), puoi farla esplodere e cancellare questa attività in modo che non venga più eseguita. Se invece la mucca è in volo, devi accertarti che sia ancora in fiamme con `cow.setFireTicks(600)` e spingere il suo livello di salute al massimo per farla vivere ancora un po'. (È più crudele scriverlo che farlo…)

Il risultato? Muovi il cuoio come una bacchetta magica e lancia mucche in fiamme che esplodono all'impatto con il terreno.

Il codice completo per `CowShooter.java` e `CowTask.java` è in *code/CowShooter/src/cowshooter*.

È molto più divertente di `Hello, World`, vero?

Per continuare

In questo capitolo hai visto alcuni meccanismi di Java, compreso quello per creare una nuova classe, e hai imparato qualcosa in più sulla creazione di oggetti dalle classi. La parte divertente, però, è poter eseguire ininterrottamente le attività sullo sfondo, implementando cose interessanti come mucche in fiamme che esplodono.

Nelle prossime pagine vedremo come salvare i dati e tenerne traccia anche dopo che il gioco è stato interrotto, ricaricato o chiuso.

La tua toolbox

68%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.
- Usare gli oggetti Java.
- Usare le istruzioni `import` per i package Java.
- Usare `new` per creare gli oggetti.
- Aggiungere un nuovo comando a un plug-in.
- Lavorare con gli oggetti `Location`.
- Trovare blocchi ed entità.
- Usare le variabili locali.
- Usare le variabili globali a livello di classe.
- Usare gli `ArrayList`.
- Usare le `HashMap`.
- Usare `private` e `public` per controllare la visibilità.
- Modificare i blocchi di Minecraft.
- Modificare e generare le entità.
- Monitorare gli eventi di gioco e rispondere.
- Gestire i permessi dei plug-in.
- Creare una classe separata.
- Programmare un'attività che deve essere svolta nel futuro.

- Programmare un’attività in modo che venga svolta in continuazione.

File di configurazione e dati di gioco

Siamo arrivati a un punto. Grazie a quanto hai imparato finora, sei in grado di ascoltare gli eventi, programmare le attività, conservare i dati in liste o hash e farli usare al tuo plug-in finché il server rimane operativo. Sei quindi in una buona posizione per realizzare dei plug-in accattivanti.

Tuttavia finora abbiamo trascurato un aspetto importante: i server non sono sempre al lavoro.

Qualsiasi server di Minecraft (compreso il tuo) a un certo punto può arrestarsi. Tu stesso l'hai fermato diverse volte seguendo gli esempi di questo libro per installare i plug-in. Ma un server può bloccarsi anche perché l'hardware si rompe, o magari semplicemente perché hai spento il computer. E quando il server si ferma, il tuo plug-in dimentica tutto quello che sapeva, e i dati vanno perduti.

Ci occorre allora un modo per memorizzare i dati importanti da qualche parte sul disco, dove poterli salvare spesso e da cui ricaricarli in caso di necessità.

I dati di cui prendersi più cura sono di due tipi.

- I dati di configurazione, che contengono tutto quello che devi sapere su come funziona il plug-in. In genere vengono impostati una sola volta e aggiornati raramente, ma sono sempre pronti quando il plug-in è attivo. Probabilmente li hai già utilizzati in altri plug-in o nell'impostazione del server per lo stesso Minecraft.
- I dati di gioco, che contengono cose come i punteggi dei giocatori, l'inventario e altri elementi che cambiano spesso durante il gioco. Sono i più insidiosi, come scopriremo tra poco.

Vediamo nel dettaglio questi tipi di dati e come salvarli e caricarli.

Usare un file di configurazione

Molti plug-in, e anche il server, utilizzano *file di configurazione* per ritoccare e personalizzare i valori e per selezionare comportamenti differenti. Un file di configurazione è un semplice file di testo concepito per essere letto e modificato direttamente da una persona. L'idea è che puoi personalizzare gli aspetti del plug-in senza dover impazzire con il codice sorgente o ricompilare qualcosa. Utenti finali, amministratori del server, sviluppatori: tutti possono utilizzarli. Lo faremo anche noi.

In particolare, useremo un file di configurazione per modificare un plug-in. Nel Capitolo 6 abbiamo costruito il plug-in `SquidBomb`, che faceva cadere dei calamari sopra alla nostra posizione corrente (i calamari-bomba). Aggiungeremo al plug-in le seguenti opzioni di configurazione:

- il numero di calamari da far cadere;
- l'altezza da cui cadranno i calamari;
- la possibilità di eliminare un calamaro uccidendolo o dandogli fuoco (una novità!).

Per nostra fortuna, i plug-in di Canary sono dotati di un meccanismo per il file di configurazione già pronto all'uso. Ecco come funziona.

Il file di configurazione si chiamerà *NomePlugin*`.cfg` e verrà collocato in una sottocartella che ha lo stesso nome del plug-in, sotto a *Server/config*. Per il nostro plug-in `SquidBombConfig`, il file di configurazione si trova qui:

```
~/Desktop/server/config/SquidBombConfig/SquidBombConfig.cfg
```

e appare così:

```
numSquids=6 squidDropHeight=5.0 setFire=false
```

Il formato usato nel file è molto semplice, e già lo conosci, perché è lo stesso del file `Canary.inf`.

Ogni riga rappresenta un'impostazione e contiene solo il nome del valore che dovrà essere memorizzato (ricorda di non inserire spazi nel nome), seguito da un segno di uguale (`=`) e poi dal valore. Puoi salvare stringhe, integer, valori booleani (vero o falso), double e così via.

Se all'inizio il plug-in non contiene ancora il file `cfg`, ne viene creato uno

predefinito in base ai valori impostati nel plug-in. È un buon punto di partenza: una volta che il file è stato generato, puoi entrarci e modificarlo sul disco (può farlo anche un utente o un operatore del server).

Studiamo come intervenire sul codice del plug-in per usare un file di configurazione. La parte importante qui è la funzione `enable()`. Non appena viene chiamata, recupera i valori dal file di configurazione e li imposta in una serie di variabili statiche che il plug-in potrà utilizzare. Di seguito vedi quello che ho aggiunto io.

SquidBombConfig/src/squidbombconfig/SquidBombConfig.java

```
private static int numSquids;
private static double squidDropHeight;
private static boolean setFire;

// Server/config/SquidBombConfig/SquidBombConfig.cfg:
//   numSquids=6
//   squidDropHeight=5
//   setFire=false

public boolean enable() {
  super.enable();//Se non lo fai tu, la chiamerà il compilatore
  logger.info("Getting config data");
  PropertiesFile config = getConfig();
  numSquids = config.getInt("numSquids", 6);
  squidDropHeight = config.getDouble("squidDropHeight",  5.0);
  setFire = config.getBoolean("setFire", false);
  config.save(); // Ne crea un altro, se serve
  return true;
}
```

Come prima cosa chiamiamo `getConfig()` per ottenere l'oggetto `PropertiesFile` per il plug-in. (Se non c'è ancora il file vero e proprio, otterremo un oggetto vuoto.) Con questo possiamo iniziare a effettuare delle chiamate per recuperare i valori dal file di configurazione.

Ogni chiamata per leggere i dati assomiglia a `config.get`*Tipo*`()`, dove *Tipo* è il tipo di variabile che stiamo cercando di ottenere, quindi `getInt`, `getDouble`, `getBoolean`, `getString` e così via.

Nota che devi usare la versione *camel-case* del tipo di dato, quindi `getInt` e non `getint`; `getBoolean` e non `getboolean`; `getDouble` e non `getdouble`...

Usare le funzioni `get`*XXX* e `set`*XXX* per leggere e salvare le variabili è un comportamento standard in Java; da qui i nomi *getter* e *setter* per questo tipo di funzioni (nomi poco originali, lo ammetto).

Per ogni getter del file di configurazione, puoi specificare un valore predefinito

da usare nel caso in cui quell'impostazione non si trovi nel file. Per esempio, chiamando

```
numSquids = config.getInt("numSquids", 6);
```

il `numSquids` (cioè il numero di calamari) verrà impostato a 6, anche se l'impostazione per `numSquids` non si trova nel file di configurazione o se, addirittura, il file di configurazione non esiste. Se nel file è presente un'impostazione, allora `numSquids` verrà impostato a quel valore.

Al termine della funzione chiameremo `save()`. Se il file di configurazione ancora non esiste, la chiamata lo creerà con i valori predefiniti. Altrimenti, se non abbiamo cambiato alcun valore, non accadrà niente.

Più avanti nel plug-in, useremo le variabili statiche a livello di classe invece dei numeri che abbiamo inserito direttamente nella versione precedente.

SquidBombConfig/src/squidbombconfig/SquidBombConfig.java

```
double y = loc.getY();
loc.setY(y + squidDropHeight);
me.chat("Spawning " + numSquids + " squid.");
// Genera alcuni calamari. Bella roba.
for (int i = 0; i < numSquids; i++) {
  spawnEntityLiving(loc, EntityType.SQUID);
}
```

Un gioco da ragazzi.

Prova da solo

Esegui `build.sh` nel tuo nuovo plug-in `SquidBombConfig` e prova il comando `squidbombc`: dovrebbe funzionare esattamente come nella versione già vista (anche se questa versione ha una "c" alla fine, quindi è `squidbombc`).

Se accedi alla cartella di configurazione del tuo server, adesso vedrai una nuova cartella *SquidBombConfig*. Al suo interno troverai il nuovo file `SquidBombConfig.cfg`:

```
~/Desktop/server$ cd config
~/Desktop/server/config$ ls
SquidBombConfig       motd.txt              plugin_priorities.cfg worlds
db.cfg                ops.cfg               server.cfg
~/Desktop/server/config$ cd SquidBombConfig/
~/Desktop/server/config/SquidBombConfig$  ls
SquidBombConfig.cfg
~/Desktop/server/config/SquidBombConfig$  cat SquidBombConfig.cfg
numSquids=6
squidDropHeight=5.0
setFire=false
```

Qui ho utilizzato il comando `cat` per eliminare il contenuto del file.

Modifica il file e cambia il numero di calamari in qualcosa di più grande, per esempio 12. Salva il file e riavvia il server.

Adesso, quando eseguirai il comando `SquidBomb`, sarai sommerso di calamari.

Congratulazioni! Ora i tuoi utenti potranno ritoccare il plug-in senza nemmeno sfiorare il codice sorgente.

Tutti questi calamari che si accumulano creano un po' di disordine, quindi dobbiamo fare pulizia. Per lanciare la Grande Purga dei Calamari ho aggiunto un comando `squidpurge` al plug-in (vedremo il sorgente tra poco). Provalo.

Oltra a limiti e quantità, puoi modificare anche il comportamento del plug-in. Torna al file `SquidBombConfig.cfg` e porta `setFire` a `true`. Riavvia il server, e quando deciderai di eliminare tutti i calamari, darai loro fuoco invece di ucciderli semplicemente. Una dozzina di calamari in fiamme!

Plug-in: SquidBombConfig

Diamo un’occhiata al codice completo e finale del plug-in `SquidBombConfig`.

SquidBombConfig/src/squidbombconfig/SquidBombConfig.java

```
package squidbombconfig;

import java.util.Collection;
import java.util.Iterator;

import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.api.entity.EntityType;
import net.canarymod.api.entity.living.EntityLiving;
import net.canarymod.api.entity.living.animal.Squid;
import com.pragprog.ahmine.ez.EZPlugin;
import net.visualillusionsent.utils.PropertiesFile;

public class SquidBombConfig extends EZPlugin {
  private static int numSquids;
  private static double squidDropHeight;
  private static boolean setFire;

  // Server/config/SquidBombConfig/SquidBombConfig.cfg:
  //    numSquids=6
  //    squidDropHeight=5
  //    setFire=false

  public boolean enable() {
    super.enable();//Se non lo fai tu, la chiamerà il compilatore
    logger.info(“Getting config data”);
    PropertiesFile config = getConfig();
    numSquids = config.getInt(“numSquids”, 6);
    squidDropHeight = config.getDouble(“squidDropHeight”,  5.0);
    setFire = config.getBoolean(“setFire”, false);
    config.save(); // Ne crea un altro, se serve
    return true;
  }

  @Command(aliases = { “squidbombc” },
            description = “Drop a configurable number of squid on your head.”,
            permissions = { “” },
            toolTip = “/squidbombc”)
  public void squidbombCommand(MessageReceiver caller, String[] args) {
    if (caller instanceof Player) {
    Player me = (Player)caller;
    Location loc = me.getLocation();
    double y = loc.getY();
    loc.setY(y + squidDropHeight);
    me.chat(“Spawning “ + numSquids + “ squid.”);
    // Genera alcuni calamari. Bella roba.
    for (int i = 0; i < numSquids; i++) {
        spawnEntityLiving(loc, EntityType.SQUID);
    }
  }
```

```
}

➢   @Command(aliases = { "squidpurge" },
➢             description = "Get rid of squid.",
➢             permissions = { "" },
➢             toolTip = "/squidpurge")
➢   public void squidpurgeCommand(MessageReceiver  caller, String[] args) {
➢     if (caller instanceof Player) {
➢        Player me = (Player)caller;
➢
➢        Collection<EntityLiving> squidlist = me.getWorld().getEntityLivingList();
➢        for (EntityLiving entity : squidlist) {
➢          if (entity instanceof Squid) {
➢            Squid victim = (Squid)entity;
➢            if (setFire) {
➢              victim.setFireTicks(600);
➢            } else {
➢              victim.setHealth(0.0f);
➢            }
➢          }
➢        }
➢      }
➢    }
  }
```

Le frecce ➢ indicano dove ho aggiunto il comando `squidpurge`. Ottengo una lista di tutti i calamari presenti nel mondo con `me.getWorld().getEntityLivingList()` e poi percorro la lista con un iteratore. Per ogni calamaro, a seconda dell'impostazione nel file di configurazione, posso decidere di dargli fuoco, così:

```
victim.setFireTicks(600);
```

oppure posso semplicemente ucciderlo impostando la sua salute direttamente a zero:

```
victim.setHealth(0.0f);
```

Bastano una piccola istruzione `if` e un file di configurazione per consentire agli utenti di perfezionare il comportamento del plug-in senza intervenire sul codice sorgente. Fantastico.

Memorizzare i dati di gioco in un database

Tutto questo è eccezionale per i dati di configurazione, ma non è così utile per i dati di gioco, che cambiano man mano che il gioco procede, come punteggi e stato dei giocatori, inventario, salute e via di seguito. Dovremo accedere a queste informazioni in modo leggermente diverso, e dovremo riuscire a scriverle dal plug-in mentre questo le legge.

Quando si parla di salvataggio, tuttavia, dobbiamo affrontare una limitazione notevole. Si possono salvare e caricare solo i tipi semplici, come stringhe, interi, decimali e valori booleani. Non si possono invece salvare oggetti specifici di Minecraft, come una `Location` o un `Player`. Si può salvare il nome di un giocatore come `String` e una `Location` come un insieme `double` delle coordinate x, y e z, ma non si possono salvare direttamente gli oggetti di Minecraft.

In realtà questa è una limitazione fino a un certo punto, perché non dovresti comunque salvare oggetti `Player`, mondi, posizioni o altri elementi “vivi” del gioco.

Ricorda che Minecraft si gioca in tempo reale. I giocatori si disconnettono, cambiano posizione, i blocchi cambiano tipo, le cose prendono fuoco, cadono ed entrano o escono dall’inventario, tutto mentre il codice è in esecuzione. Se salvi un oggetto `Player` su disco, o anche in un elenco in memoria, non hai la garanzia che sarà ancora utilizzabile così com’è il giorno o la settimana dopo. Anzi, è molto probabile che non lo sia. Basta che un giocatore si disconnetta e zac!, il vecchio oggetto `Player` non è più valido.

Invece di salvare un oggetto `Player`, allora, salva il nome del giocatore. Quando caricherai nuovamente i dati nel gioco per utilizzarli, assicurati che il giocatore sia effettivamente online.

Ci sono due modi per salvare e caricare i dati di gioco attraverso Canary. Uno è ricorrere a un database SQL completamente sviluppato, come SQLite o MySQL, usando il linguaggio SQL per cercare e memorizzare i dati. È però un metodo complesso, e qui non ho lo spazio per spiegartelo bene. Tuttavia, se hai a che fare con migliaia di utenti su un grosso server e con insiemi complicati di dati annidati, puoi affrontare le tecniche SQL. Trovi un tutorial ufficiale online all’indirizzo

http://bit.ly/1zZrsPZ, ma è molto avanzato, e può creare confusione, anche nei professionisti.

Se invece le tue esigenze per quanto riguarda i dati sono relativamente semplici e hai a che fare con poche centinaia di utenti, allora Canary ha quello che fa per te: gli oggetti `DataAccess` e il `Database` per gestirli.

Quado si lavora con i database, per descrivere le cose si usano parole leggermente diverse da quelle che si usano nel codice. Nel codice puoi avere una classe, un record (costituito da variabili) o dei campi. I database hanno gli stessi elementi, ma i campi vengono chiamati *colonne*, e le colonne vengono raggruppate in una *tabella.*

Visto che la ricerca dei record è un'operazione fondamentale in un database, in ogni tabella abbiamo a disposizione un *campo della chiave primaria*, dedicato a quanto cerchiamo più spesso. "Chiave primaria" significa che in quella tabella c'è un solo record che ha quel valore. Per esempio, `player_name` è un'ottima chiave primaria: visto che ogni giocatore ha un nome univoco, costituisce un metodo rapido per trovare un giocatore specifico.

Oggetti DataAccess

Nell'API Canary puoi creare un oggetto `DataAccess` che definisce quello che vuoi salvare in un database. Per esempio, immagina di voler memorizzare la posizione di un giocatore. Dovrai poterla salvare e cercare quel record in base al nome del giocatore, e magari salvare anche le coordinate x, y e z.

Per farlo, inizia con una classe separata, tratta da `DataAccess`, che contiene solo i dati che vuoi caricare e salvare:

```
public class AllPlayerLocations extends DataAccess {
  public String player_name;
  public double x;
  public double y;
  public double z;
}
```

Questi sono i dati, ma a Canary servono altre cose prima che tu possa utilizzarli in un database.

La prima cosa che vuole sapere è come chiamare questo campo nel database e quale tipo di dati rappresenta. Per ognuna delle coordinate x, y e z, denomineremo il campo del database come la variabile. Ognuno di questi valori è un `double`.

Specificherai i dettagli usando l'annotazione `@Column`, così:

```
@Column(columnName = "x",
        dataType = DataType.DOUBLE)
public double x;
```

Ora il sistema saprà come salvare ogni coordinata. Possiamo fare la stessa cosa per il nome del giocatore:

```
@Column(columnName = "player_name",
        dataType = DataType.STRING)
public String player_name;
```

specificando questa volta che è un tipo `String`. Per `player_name` dovremo però fare un'aggiunta. Vogliamo che questo campo nel database sia univoco, cioè sia la chiave primaria. Nel database dovrebbe esserci solo un record con una coordinata x, y e z per ciascun nome di giocatore; specifica il record come `columnType`:

```
@Column(columnName = "player_name",
        columnType = Column.ColumnType.PRIMARY,
        dataType = DataType.STRING)
public String player_name;
```

Ora che le colonne sono specificate, dobbiamo assegnare un nome alla tabella che conterrà questi record. Lo facciamo nel costruttore:

```
public AllPlayerLocations() {
  super("all_player_locations");
}
```

Queste righe dicono al database che vuoi salvare questi campi nella tabella chiamata `all_player_locations`.

Per finire, devi far sì che questa classe estenda la classe genitore `DataAccess` e aggiungere un metodo chiamato `getInstance` che restituisca un oggetto di questa classe. La classe `DataAccess` completa appare come segue.

LocationSnapshot/src/locationsnapshot/AllPlayerLocations.java

```
package locationsnapshot;

import net.canarymod.database.Column;
import net.canarymod.database.Column.DataType;
import net.canarymod.database.DataAccess;

public class AllPlayerLocations extends DataAccess {
  @Column(columnName = "player_name",
          columnType = Column.ColumnType.PRIMARY,
          dataType   = DataType.STRING)
  public String player_name;

  @Column(columnName = "x", dataType = DataType.DOUBLE)
  public double x;
  @Column(columnName = "y", dataType = DataType.DOUBLE)
  public double y;
```

```
  @Column(columnName = “z”, dataType = DataType.DOUBLE)
  public double z;

  public AllPlayerLocations() {
  super(“all_player_locations”);
  }

  public DataAccess getInstance() {
    return new AllPlayerLocations();
  }
}
```

Vediamo come utilizzare questo codice con un plug-in.

Plug-in: LocationSnapshot

Quello che segue è un esempio dell'uso di un oggetto `DataAccess` (in questo caso `AllPlayerLocations`) in un nuovo plug-in, `LocationSnapshot`. Forniremo due nuovi comandi:

- `/savelocations`
- `/loadlocations`

Potresti usare questo tipo di funzionalità in una gara, al termine della quale riporterai tutti i giocatori ai loro punti di partenza.

Il comando `savelocations` salva su disco le posizioni correnti di tutti i giocatori online; `loadlocations`, invece, li legge a ritroso e teletrasporta ognuno di essi nelle posizioni salvate. Utilizzeremo le funzioni del `Database di` Canary con l'oggetto `AllPlayerLocations DataAccess` per salvare e caricare un hash di `Player` e `Location`.

Questo plug-in è costituito da due parti principali: `saveLocations` e `loadLocations`. Per mantenere tutto in ordine e iniziare ad abituarci a fare le cose per bene, inseriremo il codice per ciascuna nella relativa funzione.

savelocations

La logica di `savelocations` è mostrata di seguito.

LocationSnapshot/src/locationsnapshot/LocationSnapshot.java

```
private void saveLocations() {
  List<Player> playerList = Canary.getServer().getPlayerList();
  // Per tutti i giocatori...
  for (Player player : playerList) {
    // Salva le coordinate grezze, non la Location
    AllPlayerLocations apl = new AllPlayerLocations();
    apl.player_name = player.getDisplayName();
    Location loc = player.getLocation();
      apl.x = loc.getX();
      apl.y = loc.getY();
      apl.z = loc.getZ();

 1    HashMap<String, Object> search = new HashMap<String, Object>();
      search.put("player_name", player.getDisplayName());

      try {
 2      Database.get().update(apl, search);
      } catch (DatabaseWriteException e) {
        logger.error(e);
        logger.info("error");
      }
```

```
    }
  }
```

Qui percorriamo l’elenco di tutti i giocatori online. Inseriamo poi il nome e le coordinate x, y e z di ciascuno di essi in un nuovo oggetto `AllPlayerLocations`. Questo verrà salvato nel database con la chiamata a `update()` al punto **2**.

Come fa il database a sapere cosa dev’essere aggiornato? Dobbiamo passargli un’`HashMap` che specifica il record che vogliamo aggiornare (o crearla, se è la prima volta). Al punto **1**, abbiamo creato un hash che dice al database “aggiorna il record in cui il campo `player_name` è uguale al `getDisplayName()` di questo giocatore”.

Ora possiamo chiamare la funzione di aggiornamento del database:

```
try {
  Database.get().update(apl, search);
} catch (DatabaseWriteException e) {
  logger.error(e);
  logger.info(“error”);
}
```

C’è qualcosa che però non abbiamo ancora spiegato; cosa fa tutto questo misterioso codice aggiuntivo?

Catturare le eccezioni in Java

Vedi il blocco di codice racchiuso tra le parole chiave `try`/`catch`? Le funzioni del database sono dichiarate in modo da “rilevare delle eccezioni” quando qualcosa va storto (per esempio, potrebbe esserci stato un errore di scrittura del file perché il disco è pieno).

Un’*eccezione* interrompe il codice in esecuzione, salta il resto della funzione e passa direttamente al blocco `catch`. È come se in questo momento si mettesse a suonare un allarme antincendio e tu ti alzassi e corressi fuori, senza finire di leggere la pagina. Se non ci sono errori, invece, il blocco `catch` non viene mai eseguito.

Per semplificare, il nostro codice cattura l’eccezione `DatabaseWriteException` e registra l’errore, ignorandolo per il resto. Questo è un modo un po’ “dilettantesco” di gestire le eccezioni, però, e non sempre è possibile ignorarle. La domanda che dovremmo farci invece è “di chi è il problema, e chi può correggerlo?”.

Solitamente il codice che chiami in una libreria non ha la minima idea del motivo per cui lo stai usando, per cui, quando qualcosa non va, non immagina cosa

intendi fare per gestire il problema. Vuoi che si blocchi tutto il server? Vuoi che venga restituito un errore all'utente? Vuoi modificare alcune variabili e annullare quello che stavi cercando di fare? Poiché il codice della libreria non lo sa, non può effettuare nessuna correzione. L'unica cosa che può fare è sollevare un'eccezione (cioè, alza le mani e si arrende).

Quando si tratta di eccezioni, hai due possibilità. Puoi affrontare direttamente gli errori e gestire e catturare l'eccezione da solo, oppure puoi arrenderti anche tu e passare l'eccezione al chiamante. In questo secondo caso, dichiari che la tua funzione `throws DatabaseWriteException` (cioè "rileva questa eccezione o un qualsiasi altro tipo"). Non ti serve nessun `try`/`catch`: adesso è diventato un problema di qualcun altro.

In teoria, le eccezioni dovrebbero essere utilizzate solo per situazioni… eccezionali, appunto. Se intendevi creare una mucca e invece ottieni un Creeper, questa è una situazione eccezionale: di più, è un disastro non previsto. Se hai appena aggiunto un giocatore a un elenco di giocatori e l'elenco risulta ancora a zero, è un altro disastro.

Provare a cercare qualcosa nel database e non trovarlo non è però così drammatico, anzi, accade spesso. In casi come questo potresti voler catturare l'eccezione e ignorarla.

E cosa succede se non riesci a scrivere i dati nel database, come stiamo facendo qui? Il problema potrebbe essere più serio: magari si è esaurito lo spazio su disco sul server, tanto per fare un esempio. Cosa fare?

Personalmente, cercherei di arrestare il server nel modo meno traumatico possibile. Un programma che viene chiuso fa molti meno danni di un programma che si interrompe durante l'esecuzione. Quando si scrive codice per i sistemi più professionali, si impara presto che questo è il modo migliore di procedere: trovare il problema il prima possibile, fermare tutto e segnalare chiaramente che qualcosa non va come dovrebbe.

Non è detto che l'inconveniente si verifichi sempre sul nostro server; abbiamo appena creato un plug-in eseguito da altri, e non sarebbe educato fermare il loro server solo perché non riusciamo a scrivere un file. In questo caso registreremo un messaggio nella console di Minecraft per segnalare il problema e continueremo a lottare.

loadlocations

Proseguiamo. Quando digiti il comando `loadlocations`, percorri la lista dei giocatori che sono online e recuperi dal database la loro posizione salvata; poi converti le coordinate x, y e z nella `Location` opportuna e teletrasportali lì.

LocationSnapshot/src/locationsnapshot/LocationSnapshot.java

```
private void loadLocations() {
  //Percorri la lista dei giocatori; se si trovano nell'hash, teletrasportali.
  List<Player> playerList = Canary.getServer().getPlayerList();
  for (Player player : playerList) {
    String name = player.getDisplayName();

    AllPlayerLocations apl = new AllPlayerLocations();
    HashMap<String, Object> search = new HashMap<String, Object>();
    search.put("player_name", name);

    try {
      Database.get().load(apl, search);
    } catch (DatabaseReadException e) {
      logger.info(name + " is not online");
      continue;
    }

    // Ricostruisci una Location partendo dalle coordinate
    Location loc = new Location(apl.x, apl.y, apl.z);
    logger.info("Teleporting " + name + " to " + printLoc(loc));
    player.teleportTo(loc);
  }
}
```

Ovviamente il salvataggio deve essere effettuato percorrendo l'elenco di tutti i giocatori online, ed è facile farlo anche per il caricamento. Potremmo anche iniziare da una lista di tutti i giocatori che abbiamo salvato nel database e poi verificare se sono online, ma l'altro metodo è più semplice. In entrambi i casi, potrebbero esserci giocatori che prima erano online e adesso non lo sono più, o viceversa.

Dobbiamo allora trovare questo giocatore nel database, utilizzando una mappa di ricerca e la funzione `load`. Quando saremo pronti per treletrasportarlo, creeremo un nuovo oggetto `Location` basandoci sulle coordinate che abbiamo salvato e lo porteremo lì.

Compila e installa `LocationSnapshot` e fai una prova. Connettiti dal tuo client ed esegui il comando `/savelocations`.

Esci dal client, arresta il server e riavvia tutto. Spostati da qualche altra parte nel gioco e prova il comando `/loadlocations`. Verrai riportato alla posizione salvata (così come qualsiasi altro giocatore sul server).

XML O SQLITE

Nel file di configurazione del server, `server/config/server.cfg`, puoi scegliere il sistema che il database utilizzerà per salvare e caricare. Di solito si usa un file nel formato XML, che verrà salvato nella cartella *server/db*.

Puoi cambiarlo in modo che utilizzi `xml`, `mysql` o `sqlite` modificando questa riga nel file di configurazione del server:

```
data-source=xml
```

Per ottenere prestazioni migliori, puoi modificare questa opzione in `sqlite` senza compiere ulteriori configurazioni. Per utilizzare l'opzione `mysql`, dalle prestazioni ancora più eccezionali, impostala usando `server/config/db.cfg` ed eseguendo un server MySQL.

Plug-in: BackCmd con funzioni di salvataggio

Applicando gli stessi concetti a `LocationSnapshot`, aggiungeremo adesso la funzionalità di salvataggio dei dati al plug-in `BackCmd`, in modo da poter recuperare le posizioni che abbiamo salvato a ogni riavvio del server. Io ti fornirò la struttura e il tipo di funzioni, ma sarai tu a dover scrivere il corpo delle funzioni, cioè il codice vero e proprio che fa girare il plug-in. Allaccia le cinture!

`BackCmd` è un plug-in è un po' più complesso di `LocationSnapshot`. Come forse ricorderai dal Capitolo 9, `BackCmd` tiene traccia di uno stack di oggetti `Location` per ciascun giocatore, e non solo di una singola posizione.

Ricorda che il sistema del database non è in grado di memorizzare oggetti `Location`; può memorizzare una lista, che possiamo usare come stack, ma questa può essere solo di un tipo di base. Non può salvare un elenco di coordinate x, y e z né un elenco di array. Qui imbroglieremo un po': salveremo le coordinate in una stringa in fase di scrittura e poi le recupereremo da una stringa sotto forma di double distinti in fase di lettura.

Ecco la procedura da seguire per il codice da scrivere.

1. Crea una `SavedLocation`, che sarà un oggetto `DataAccess`.
2. Il plug-in principale lavora con i giocatori, quindi aggiungeremo una nuova `Location` e recupereremo l'ultima posizione di un giocatore specifico. Ha quindi senso che `SavedLocation` appaia come uno stack, con un metodo `pop` e un metodo `push`.
3. Poiché il plug-in vedrà `SavedLocation` come uno stack, eseguirà internamente (come funzioni private) le operazioni di salvataggio e caricamento nel database. In `SavedLocation`, dovrai scrivere le funzioni private che chiameranno il database e controlleranno le eventuali eccezioni.
4. Scrivi una funzione di conversione che prende un oggetto `Location` e restituisce una stringa che contiene ogni coordinata separata da virgole (`,`).
5. Scrivi una funzione di conversione che prende una stringa di coordinate separate da virgole (`,`) e costruisce un oggetto `Location`.

Questo è solo il primo passo da compiere. Man mano che procederemo, salteranno fuori altre cose, ma è giusto, perché è così che funziona lo sviluppo software.

Con il tempo, inoltre, non temere di aggiungere dei messaggi `logger.info()` come aiuto per seguire quanto sta facendo il plug-in.

Abbiamo fatto il pieno. Partiamo.

Creare una classe SavedLocation

Poiché non puoi salvare uno stack di oggetti `Location` direttamente sul disco, dovrai creare un oggetto che invece puoi memorizzare, cioè un `DataAccess`. Qui creerai una `SavedLocation` che salverà una lista di stringhe per ciascun giocatore.

Cominciamo con lo scrivere un po' di codice. Entra nella cartella *src/backcmd* del plug-in `BackCmd` e crea un nuovo file chiamato `SavedLocation.java`. La prima cosa che ti serve è un'istruzione del package e qualche istruzione di importazione.

L'inizio della classe `SavedLocation` è simile al seguente.

BackCmdSave/src/backcmdsave/SavedLocation.java

```
package backcmdsave;

import java.util.ArrayList;
import java.util.HashMap;
import net.canarymod.database.Column;
import net.canarymod.database.Column.DataType;
import net.canarymod.database.DataAccess;
import net.canarymod.database.exceptions.*;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.World;
import net.canarymod.database.Database;
import net.canarymod.Canary;

public class SavedLocation extends DataAccess {
```

Copia questo codice nel tuo nuovo file o digitalo man mano.

A seguire dovrai aggiungere le annotazioni `@Column` per descrivere i campi che vuoi salvare nel database. Ricorda che ti occorrono due campi:

- `player_name`: una stringa, e una chiave primaria, proprio come abbiamo visto nel plug-in `LocationSnapshot`;
- `location_strings`: un elenco di stringhe. Per creare una lista per un campo, aggiungi il parametro `isList = true` all'annotazione e rendi la variabile un

elenco di stringhe, scrivendo `ArrayList<String>` invece che semplicemente `String`.

Fai una prova. Otterrai qualcosa simile a quanto riportato di seguito.

BackCmdSave/src/backcmdsave/SavedLocation.java

```
@Column(columnName = "player_name",
  columnType = Column.ColumnType.PRIMARY,
  dataType = DataType.STRING)
public String player_name;

@Column(columnName = "location_strings",
  dataType = DataType.STRING,
  isList = true)
public ArrayList<String> location_strings;
```

Prima di compilare questo codice ti servono altre due o tre cose, come abbiamo visto nel plug-in `SavedLocation`.

1. Aggiungi un costruttore predefinito che fornisca un nome per questa tabella nel database (chiamiamola `"saved_player_locations"`).
2. Aggiungi una funzione chiamata `getInstance` che restituisca un oggetto `DataAccess` come una nuova istanza.
3. Aggiungi un altro costruttore che prenda una stringa per il nome del giocatore. Nel corpo del costruttore devi chiamare `super` proprio come nel costruttore predefinito, e poi effettuare l'assegnazione `player_name`.

Al termine, dovresti ottenere questo risultato.

BackCmdSave/src/backcmdsave/SavedLocation.java

```
public SavedLocation() {
  super("saved_player_locations");
}

public SavedLocation(String name) {
  super("saved_player_locations");
  player_name = name;
}

public DataAccess getInstance() {
  return new SavedLocation();
}
```

Compilalo usando `build.sh`, come sempre.

Leggere dal database e scriverci dentro

Per quanto ne sa il database, stiamo lavorando con due soli campi, `player_name` e `location_strings`, che usano `player_name` come chiave.

Tornando a `LocationSnapshot`, puoi copiare le funzioni `load` e `update` del database, con la gestione delle eccezioni, e creare qui due funzioni:

```
private void myRead(final String name) {
}

private void myWrite() {
}
```

In ciascuna funzione, impostiamo l'`HashMap` per la ricerca, poi effettuiamo il `load` (il caricamento) o l'`update` (l'aggiornamento) del database in un blocco `try/catch`.

Fai una prova. Il risultato dovrebbe essere il seguente.

BackCmdSave/src/backcmdsave/SavedLocation.java

```
private void myRead(final String name) {
  player_name = name;

  HashMap<String, Object> search = new HashMap<String, Object>();
  search.put("player_name", name);

  try {
    Database.get().load(this, search);
  } catch (DatabaseReadException e) {
    // Non è per forza un errore, potrebbe essere il primo
  }
  if (location_strings == null) {
    player_name = name;
    location_strings = new ArrayList<String>();
  }
}


private void myWrite() {
  HashMap<String, Object> search = new HashMap<String, Object>();
  search.put("player_name", player_name);

  try {
    Database.get().update(this, search);
  } catch (DatabaseWriteException e) {
    //Errore, non è possibile scrivere!
    System.err.println("Update failed");
  }

}
```

Compilalo usando `build.sh`, come al solito.

Ora abbiamo una classe `DataAccess` che effettuerà gli aggiornamenti e caricherà dal database. Tuttavia, non disponiamo ancora di un modo valido per connetterla al plug-in. Ce ne occuperemo ora.

Trasformare le posizioni in stringhe e viceversa

Prima di continuare, dobbiamo affrontare la conversione delle funzioni per una

`Location`, senza la quale non potremo iniziare a lavorare sugli stack.

Convertire una `Location` in una stringa è semplice: costruisci una stringa per ognuno dei risultati di `getX()`, `getY()` e `getZ()` usando un carattere non numerico per separare ciascun valore; potrebbe essere un carattere di spazio, ma personalmente preferisco utilizzare le virgole (`,`), che mi sembra facilitino la lettura.

Scrivi una funzione chiamata `locationToString` che prende una `Location` e restituisce una `String`.

E per il contrario? Avendo per esempio una stringa come `"110,75,220"`, come si fa a dividerla in tre parti, una per ciascun valore?

Java ti viene in soccorso. Nella documentazione relativa alle stringhe troverai una funzione chiamata `split`, che prende una stringa e la suddivide in un array. Per esempio, partendo dalla stringa `str="110,75,220"`, la funzione `str.split(",");` restituirà un array di tre stringhe: "110", "75" e "220". Dopodiché è sufficiente utilizzare `Double.parseDouble(string)` per convertire ciascuna stringa in un double, e con i double creare una nuova `Location`.

Ora puoi scrivere una funzione `stringToLocation` che prende una `String` e restituisce una `Location`. Il risultato finale dovrebbe assomigliare al seguente.

BackCmdSave/src/backcmdsave/SavedLocation.java

```
private String locationToString(Location loc) {
  return loc.getX() + "," +
         loc.getY() + "," +
         loc.getZ();
}

private Location stringToLocation(String str) {
  String[] arr = str.split(",");
  double x = Double.parseDouble(arr[0]);
  double y = Double.parseDouble(arr[1]);
  double z = Double.parseDouble(arr[2]);
  return new Location(x, y, z);
}
```

Assicurati che si compili usando il solito `build.sh`.

Agire come uno stack

Sebbene questo sia un oggetto `DataAccess`, siamo liberi di aggiungere a esso altre funzioni e dati per facilitare la costruzione del plug-in. Dal punto di vista del plug-in principale, dovremo riuscire a trattare una `SavedLocation` (cioè una posizione

salvata) di un giocatore specifico come uno stack di `Location`. Questo significa che dovremo esporre almeno una funzione `push` e `pop` pubblica. Troviamo subito un ostacolo: `location_strings` è dichiarata come un’`ArrayList<String>`, e non come uno `Stack`. Come è possibile, allora, implementare `push` e `pop` usando una normale lista?

La parte relativa al `push` è semplice: un metodo `add` aggiungerà una nuova voce alla fine della lista, che è quella che utilizzerai per il `push`.

Per quanto riguarda il `pop`, invece, dovrai `remove` (eliminare) l’ultima voce nella lista usando il suo indice, cioè la `size()` (la dimensione) della lista meno uno.

Mi pare accettabile. Nelle funzioni `push` pubbliche, devi poter leggere lo stack del giocatore dal database; chiama una funzione interna privata per inserire effettivamente quella posizione nello stack e poi salvala nel database. Il meccanismo per `pop` è analogo: leggi dal database, fai l’aggiunta allo stack e scrivi il nuovo stack nel database. Nel plug-in `BackCmd` che abbiamo visto in precedenza non era sufficiente aggiungere l’ultimo valore; avevamo dovuto utilizzare una funzione `equalsIsh` per assicurarci di essere abbastanza lontani dalla posizione in cui avveniva il teletrasporto.

BackCmdSave/src/backcmdsave/SavedLocation.java

```
public void push(Location loc) {
  myRead(player_name);
  //Garantisce che la posizione precedente è diversa, se esiste
  if (peek_stack() == null ||
    !equalsIsh(peek_stack(), loc)) {
    push_stack(loc);
    myWrite();
  }
}

public Location pop(Location here) {
  myRead(player_name);
  if (location_strings.size() == 0) {
  return null;
  }
  Location loc = pop_stack();

  myWrite();
  return loc;
}
```

Devi scrivere ancora un paio di cose: le funzioni interne `push_stack`, `pop_stack` e `peek_stack` e le funzioni del database, `myRead` e `myWrite`. Puoi copiare `equalsIsh` direttamente dal plug-in `BackCmd`.

Partiamo dalle funzioni interne dello stack. Lo scopo è quello di passare una

`Location` nello stack e ottenere indietro una `Location` con `peek` e `pop`, quindi utilizziamo le funzioni di conversione `locationToString` e `stringToLocation`.

Prima prova da solo. Al termine, torna qui per vedere se il tuo codice assomiglia al seguente.

BackCmdSave/src/backcmdsave/SavedLocation.java

```
private void push_stack(Location loc) {
  String s = locationToString(loc);
  location_strings.add(s);
}

private Location peek_stack() {
  if (location_strings.isEmpty()) {
    return null;
  }
  String s = location_strings.get(location_strings.size()-1);
  return stringToLocation(s);
}

private Location pop_stack() {
  Location loc = peek_stack();
  location_strings.remove(location_strings.size()-1);
return loc;
}
```

Accertati che si compili usando `build.sh`, come sempre.

Aggiungere le funzioni di salvataggio e caricamento a BackCmd

Ora che disponi di una `SavedLocation` che sa come inserire e rimuovere le singole posizioni di un giocatore, è venuto il momento di aggiungere il codice in `BackCmd.java`, che le chiamerà.

Apri `BackCmd.java` e inizia a effettuare le seguenti modifiche.

- In `onTeleport`, cancella tutto il codice nella sezione `else` del blocco e sostituiscile con il codice che crea una nuova `SavedLocation` e chiama la sua funzione `push` con la posizione del giocatore.
- In `backCommand`, individua la posizione dove effettuare il teletrasporto creando una nuova `SavedLocation` e chiamando la sua funzione `pop` per ottenere il valore più recente.

La nuova classe principale del plug-in è riportata di seguito.

BackCmdSave/src/backcmdsave/BackCmdSave.java

```
package backcmdsave;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Stack;
import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.position.Location;
import net.canarymod.hook.HookHandler;
import net.canarymod.hook.player.TeleportHook;
import net.canarymod.plugin.PluginListener;
import com.pragprog.ahmine.ez.EZPlugin;


public class BackCmdSave extends EZPlugin implements PluginListener {
  private static List<Player> isTeleporting = new ArrayList<Player>();

  @Override
  public boolean enable() {
    Canary.hooks().registerListener(this,  this);
    return super.enable(); // Chiama anche la versione genitore della classe.
  }

  @HookHandler
  public void onTeleport(TeleportHook event) {
    Player player = event.getPlayer();
    if (isTeleporting.contains(player)) {
      isTeleporting.remove(player);
    } else {
      SavedLocation sp = new SavedLocation(player.getName());
      sp.push(player.getLocation());
    }
  }

@Command(aliases = { "dback" },
          description = "Go back to previous places that you teleported to.",
          permissions = { "" },
          toolTip = "/dback")
public void backCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;

    SavedLocation sp = new SavedLocation(me.getName());
    Location loc = sp.pop(me.getLocation());
      if (loc != null) {
        isTeleporting.add(me);
        me.teleportTo(loc);
      } else {
        me.chat("You have not teleported yet.");

      }

    }
  }
}
```

(Qui ho rinominato il comando `dback`, che sarà quindi diverso dal `BackCmd` esistente. Anche tu puoi rinominarlo, oppure puoi usare questo al posto del

vecchio `back`.)

Compila e installa il tutto con `build.sh`, e sarai pronto.

Fai un test

Dando per scontato che tutto giri come deve, fai una prova.

1. Avvia il server.
2. Connettiti dal tuo client.
3. Teletrasportati in una nuova posizione usando il comando `/tp` o lanciando un Ender Pearl. Fallo per un po' di volte.
4. Disconnettiti e arresta il server.
5. Riavvia il server e collegati di nuovo.
6. Digita il comando `/back` nel client. La cronologia dei tuoi movimenti è stata salvata e ricaricata dal disco, quindi dovresti riuscire a tornare in ogni punto in cui è avvenuto il teletrasporto.

Prova da solo

Quello che ci servirebbe adesso è un comando `clearback` che ripulisca a ritroso la cronologia dei nostri teletrasporti. (Canary fornisce già il comando di cancellazione `clear`, quindi ci occorre qualcosa di diverso.)

Procedi e implementalo. Visto che è un comando nuovo, ti serviranno una nuova sezione `@Command` e una nuova funzione. Nel codice, tutto quello che dovrai fare è ripulire lo stack del giocatore e risalvarlo nel database.

Per continuare

È stato divertente! Ormai hai quasi finito. Questo è l'ultimo gruppo di tecniche Java che dovevi conoscere. Adesso sei in grado di salvare e caricare i dati dal database, usare le funzioni dei file Java e gestire le eccezioni.

Nel prossimo capitolo faremo qualcosa di diverso: invece di scrivere del codice, studieremo una tecnica che ci aiuta a gestire il codice che scriviamo, e vedremo come tornare a una versione precedente del codice come se premessimo un grosso pulsante *Annulla*.

Al termine, ti fornirò una panoramica completa su quanto ti occorre per progettare un plug-in da zero.

La tua toolbox

81%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.
- Usare gli oggetti Java.
- Usare le istruzioni `import` per i package Java.
- Usare `new` per creare gli oggetti.
- Aggiungere un nuovo comando a un plug-in.
- Lavorare con gli oggetti `Location`.
- Trovare blocchi ed entità.
- Usare le variabili locali.
- Usare le variabili globali a livello di classe.
- Usare gli `ArrayList`.
- Usare le `HashMap`.
- Usare `private` e `public` per controllare la visibilità.
- Modificare i blocchi di Minecraft.
- Modificare e generare le entità.
- Monitorare gli eventi di gioco e rispondere.
- Gestire i permessi dei plug-in.
- Creare una classe separata.
- Programmare un’attività che deve essere svolta nel futuro.

- Programmare un'attività in modo che venga svolta in continuazione.
- Salvare e caricare i dati di configurazione.
- Costruire un codice complesso partendo da funzioni più semplici.
- Salvare e caricare i dati di gioco.
- Usare gli oggetti `DataAccess` per lavorare con il database.
- Catturare e sollevare eccezioni in Java.

Mantenere il codice in ordine e al sicuro

Durante la scrittura del codice, potresti incappare in qualche momento frustrante: tutto funzionava bene, ma poi hai aggiunto un paio di righe in alcuni file e adesso non gira più niente. Quali modifiche hai inserito? Quale di queste ha mandato tutto all'aria? Era in questo file o in quell'altro?

Non sarebbe bello poter premere una specie di grosso "pulsante *Annulla*" per eliminare il gruppo di modifiche che ha creato problemi e tornare al codice di pochi minuti prima, quando tutto andava bene, e ricominciare?

Ho un trucchetto per te.

Esiste uno strumento chiamato Git che fa esattamente questo, e anche di più. Git agisce come un grande deposito di memoria di tutte le modifiche fatte nel codice. Puoi tornare indietro a un dato momento, anche se hai inavvertitamente eliminato o rinominato un file, se hai spostato il codice da un file a un altro e così via: Git terrà traccia di tutto per te. Inoltre puoi impostarlo in modo che una copia del suo registro possa essere copiata nel cloud, così anche se il tuo computer si rompe o se qualcuno lo ruba, il tuo codice sorgente e tutto quello che lo riguarda saranno al sicuro. Potrai poi ripristinare tutto sul tuo nuovo computer o su quello di un amico e continuare.

Non sei obbligato a usare Git, ma ti consiglio vivamente di farlo. Non è difficile da configurare, e la prima volta che ti salverà da qualche manipolazione errata dei file ti renderai conto del suo valore.

Il Web è pieno di tutorial e spiegazioni su Git, e ci sono anche un sacco di libri sull'argomento; qui faremo una veloce panoramica sui fondamenti perché tu possa iniziare a usarlo.

Installare Git

La prima cosa da fare è scaricare e installare il software di Git per Windows, OS X o Linux da `http://git-scm.com`.

Una volta che l'hai installato, probabilmente dovrai configurarlo con il tuo nome e indirizzo e-mail:

```
$ git config --global user.name "Your Name"
$ git config --global user.email "you@example.com"
```

GIT PER WINDOWS

La distribuzione Git per Windows include una versione potenzialmente valida ma anche problematica della shell bash; secondo alcuni utenti non funziona molto bene. Nell'installer di Git per Windows, seleziona *Run Git from Windows command prompt* invece dell'opzione predefinita *Git bash*.

Potresti anche incontrare inconvenienti per quanto riguarda la terminazione delle righe. Windows usa i caratteri CR LF per contrassegnare la fine delle righe, mentre altri sistemi usano LF. Java lavora bene con entrambe le convenzioni, e lo stesso la maggior parte degli editor. Tuttavia, Git potrebbe avvisarti che sta cercando di convertire i fine riga: non preoccupartene.

Git dev'essere impostato una volta per progetto. Facciamolo adesso per il plug-in `CowShooter`.

Accedi alla cartella *CowShooter* e digita `git init`:

```
$ cd CowShooter/
$ ls
Canary.inf  Manifest.txt  bin  build.sh  dist  src
$ git init
Initialized empty Git repository in /Users/andy/Desktop/code/CowShooter/.git/
```

Queste righe creano una magica cartella nascosta chiamata *.git* dove Git salverà le sue memorie: è il cosiddetto *repository*, cioè "deposito". Dovrai crearlo una volta per ciascun progetto, nel nostro caso una volta per ciascun plug-in.

Ricorda le modifiche

Ora che hai installato Git, devi dirgli quale dei tuoi file deve ricordare; non serve che siano tutti.

In genere non occorre salvare i file `.class` o `.jar`, perché questi possono essere ricreati. Invece è importante che Git ricordi tutto il tuo codice sorgente `.java`, il file di configurazione del tuo plug-in e lo script di costruzione.

Come prima cosa, ripuliamo la cartella del plug-in ed eliminiamo tutto ciò che è inutile:

```
$ rm -r bin
$ rm -r dist
```

Qui sono stati cancellati i file `.class` e `.jar`. Ti rimane un albero delle cartelle pulito con i file di cui Git deve tenere memoria insieme alle modifiche.

Usa `git add` *file...*, dove *file* può essere il nome di un singolo file o di più cartelle. Visto che per adesso vuoi che si trovi tutto nella cartella corrente, è sufficiente che digiti `git add .`:

```
$ git add .
```

(Ricorda che il carattere . indica la cartella corrente.) Ora Git sa che deve guardare questi file. Tuttavia, non dispone ancora di un'istantanea del loro stato corrente. Perché ciò avvenga, devi caricare e confermare le tue modifiche locali, cioè fare un *commit*:

```
$ git commit -a -m 'My first commit'
```

Questo comando registrerà lo stato del tuo codice sorgente nel repository di Git. `-a` indica che le modifiche verranno caricate su tutti i file di cui Git tiene traccia, mentre `-m` ti consente di specificare un *messaggio*.

Il messaggio è molto importante: attraverso di esso puoi dire cosa intendevi fare quando hai modificato questi file. Se utilizzi un messaggio del tipo "Ho fatto della roba", beh, non sarà particolarmente utile…

Facciamo qualche esperimento. Crea un nuovo file nella cartella *CowShooter* e denominalo `README.txt`. Metti nel file tutto quello che vuoi: commenti sul plug-in, il tuo manifesto personale, testo senza senso e così via.

Ora aggiungi il file e carica l'aggiunta:

```
$ git add README.txt
$ git commit -a -m "Add my personal screed"
```

```
[master 50847c8] Add my personal screed
1 file changed, 1 insertion(+)
create mode 100644 README.txt
```

Per vedere la cronologia dei messaggi di commit, si usa `git log`:

```
$ git log
commit 50847c8a3e60dbfc8f441894202765eb23fbd9a5
Author: Andy Hunt <andy@toolshed.com>
Date:   Fri Jan 7 15:51:22

    Add my personal screed

commit aa58dfa87fa79b01e2c7ce172bbcdd41b7e962d6
Author: Andy Hunt <andy@toolshed.com> Date:   Tue Jan 7 15:50:31

    First Commit
```

Queste righe mostrano le voci complete del registro, con la versione estesa dell'identificatore del commit, l'autore, la data e il messaggio. Queste stringhe apparentemente casuali (che in realtà non sono altro che hash SHA1 interessanti dal punto di vita crittografico) identificano ogni commit. Sono anche molto lunghe, però. Solitamente basta usare i primi quattro-otto caratteri dell'hash fino a produrre una stringa univoca.

Puoi ottenere un rapporto più conciso delle stesse informazioni usando queste righe:

```
$ git log --oneline
50847c8 Add my personal screed
aa58dfa First Commit
```

Di solito la versione più breve dell'hash di commit ti sarà sufficiente.

Ricorda: quando crei un nuovo file e vuoi che Git ne tenga traccia, devi aggiungerlo al repository, così:

```
$ git add myNewFile.java
```

Poi devi fare un commit per salvare un'istantanea di tutti i tuoi file nel repository. Puoi sempre controllare quali sono i file di cui Git ha conoscenza, quali no, quali sono stati caricati e quali no digitando quanto segue:

```
$ git status
# On branch master
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       bin/
#       dist/
nothing added to commit but untracked files present (use "git add" to track)
```

Accidenti! Devo aver compilato qualcosa da qualche parte, perché vedo di nuovo le cartelle `bin` e `dist`. Se questi sono i file che ti interessano, puoi effettuare l'aggiunta a Git (`git add`) in modo che possa seguirli. In questo caso però non

vogliamo che Git tenga traccia di queste cartelle, bensì che le ignori. Ecco come procedere.

1. Crea un file chiamato `.gitignore` in questa cartella (se non ce n'è già uno; `mkplugin.sh` ne genererà uno vuoto per te).
2. Nel file, inserisci i nomi dei file o delle cartelle che vuoi ignorare.
3. Aggiungi il file ed esegui il commit.

Il file `.gitignore` apparirà così:

```
bin
dist
```

Lo aggiungeremo e faremo il commit solo di quel file, specificando il nome del file invece del solito `-a`:

```
$ git add .gitignore
$ git commit .gitignore -m 'ignoring bin and dist'
[master cc0e424] ignoring bin and dist
 1 file changed, 2 insertions(+)
create mode 100644 .gitignore
```

Ecco cosa ci dice `git status` adesso:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

Perfetto! Otteniamo un messaggio rassicurante che ci dice che non c'è niente di cui fare il commit e che la cartella di lavoro è pulita. È proprio quello che si dovrebbe vedere a fine codifica.

Ora aggiungiamo del testo in fondo a `README.txt`, salviamo il file e proviamo nuovamente `git status`:

```
$ git status
# On branch master
# Changes not staged for commit:
#   (use "git add <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       modified:   README.txt
#
no changes added to commit (use "git add" and/or "git commit -a")
```

Git ha riconosciuto, intelligentemente, che hai fatto alcune modifiche al file `README.txt` e che queste non sono ancora state salvate. Un commit provvederà a farlo, e otterrai ancora il messaggio rassicurante di prima:

```
$ git status
# On branch master
nothing to commit, working directory clean
```

C'è una regola importante da ricordare: *non uscire da qui finché la cartella di*

lavoro non è pulita.

Annullare è facile

Immagina di eliminare per errore un file dalla cartella del tuo plug-in. O magari hai effettuato una serie di modifiche che si rivelano sbagliate e vuoi tornare indietro. Non aver paura: si può fare.

Nel prossimo esempio inseriremo apposta degli errori e vedremo come correggerli usando il plug-in `CowShooter` per il quale hai impostato un repository Git.

Apri `CowShooter/src/cowshooter/CowTask.java`, e in qualche punto a caso del file digita qualcosa che non ha senso, come `Zombie cows are coming! Run for your life! Braaaainsss....` ("Arrivano le mucche zombie! Mettiti in salvo! Colpisciiii....").

Java ovviamente non ha la minima idea di cosa significhi il tuo messaggio sul prossimo apocalisse delle mucche zombie. Prova a compilare con `build.sh`, e vedrai apparire un sacco di messaggi di errore come questo:

```
src/cowshooter/CowTask.java:20: illegal start of type
    Zombie cows are coming! Run for your life! Braaaainsss....
                                                             ^
src/cowshooter/CowTask.java:20: <identifier> expected
    Zombie cows are coming! Run for your life! Braaaainsss....
                                                             ^
src/cowshooter/CowTask.java:22: ';' expected
    public void run() {
          ^
src/cowshooter/CowTask.java:22: invalid method declaration;
return type required
    public void run() {
```

Arg! Qualcosa che abbiamo digitato ha messo sotto sopra la compilazione. Se ti torvassi in una situazione di emergenza autentica, non capiresti che è il falso testo di avvertimento sugli zombie che hai appena digitato.

Nasce allora una domanda interessante: cosa è cambiato localmente nei tuoi file rispetto all'ultima volta che hai fatto un commit (cioè da quando hai generato un'istantanea)?

Puoi vederlo attraverso `git diff`:

```
$ cd src
$ cd cowshooter
$ git diff CowTask.java
diff --git a/src/cowshooter/CowTask.java b/src/cowshooter/CowTask.java
index 4894201..556d0d3 100644
--- a/src/cowshooter/CowTask.java
+++ b/src/cowshooter/CowTask.java
@@ -16,6 +16,8 @@ public class CowTask extends ServerTask {
         cow = myCow;
     }
+
+Zombie cows are coming! Run for your life! Braaaainsss....
```

```
    public void run() {
      if (cow.isOnGround()) {
```

L'output ha un po' di robaccia all'inizio e poi alcune righe tratte dal cuore di `CowTask.java`, comprese due righe marcate con dei segni più (`+`): sono quelle che ho aggiunto. E sta proprio qui il problema! Inserire queste righe è stata una cattiva idea, e sarebbe bello poter tornare indietro fino all'ultimo commit e riportare il file a com'era prima.

Se non hai ancora eseguito il commit di un file, puoi sempre tornare indietro all'ultimo commit (per esempio all'ultimo punto di salvataggio) digitando questo:

```
$ git checkout MyMessedUpFile.java
```

Zitto zitto, `MyMessedUpFile.java` tornerà a essere come prima. Qualsiasi cosa che hai digitato dopo l'ultimo commit sparisce. Via. Svanita.

Nel nostro caso specifico possiamo fare così:

```
$ cd src/cowshooter
$ ls
CowShooter.java CowTask.java
$ git checkout CowTask.java
```

`CowTask.java` tornerà a uno stato conosciuto e funzionante. Possiamo anche ricompilare:

```
$ cd ~/Desktop/code/CowShooter
$ ./build.sh
Compiling with javac... Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

Disfarsi delle modifiche errate e tornare all'ultimo punto di salvataggio è facile. Ma cosa accade se hai eseguito il commit di queste modifiche un attimo prima e te ne accorgi solo ora? Puoi usare lo stesso comando, specificando però a quale commit tornare.

Proviamo. Torna in `CowTask.java` e aggiungi di nuovo le righe sbagliate sulle mucche zombie. Adesso, però, esegui il commit così:

```
$ git commit -a -m 'Added zombie warning'
[master e4ee198] Added zombie warning
 1 file changed, 2 insertions(+)
```

Possiamo fare anche di peggio. Trova la riga che dice

```
private Cow cow;
```

ed eliminala, poi esegui il commit:

```
$ git commit -a -m 'Deleted variable declaration'
[master 44b39f3] Deleted variable declaration
 1 file changed, 3 deletions(-)
```

Vediamo adesso cosa dice il log (cioè il “registro”) per `CowTask.java` (nella cartella `src/cowshooter/`):

```
$ cd ~/Desktop/code/CowShooter/src/cowshooter
$ git log --oneline CowTask.java

44b39f3 Deleted variable declaration
e4ee198 Added zombie warning
aa58dfa First Commit
```

I tuoi ID di commit, quei numeri magici che sono elencati sulla nuova riga, saranno diversi dai miei: per l’esempio usiamo i miei, però.

In questo caso vogliamo tornare all’ultima versione buona di questo file. Vogliamo cioè tornare ai commit precedenti `44b39f3` ed `e4ee198`, e riportare il file all’ID di commit `aa58dfa`, dove tutto era a posto:

```
$ git checkout aa58dfa CowTask.java
```

(Ricorda, il tuo ID di commit sarà diverso.)

`CowTask.java` tornerà allo stato precedente, prima che tu aggiungessi il testo sulle mucche zombie e che eliminassi quelle variabili.

Ora il file è effettivamente cambiato, come se lo avessi modificato a mano, quindi assicurati di caricare con un commit la modifica più recente con un messaggio opportuno (qualcosa del tipo `‘That was a bad idea’`, cioè “Non è stata una buona idea”):

```
$ git commit -a -m ‘That was a bad idea’
[master 0b6b051] That was a bad idea
1 file changed, 3 insertions(+), 2 deletions(-)
```

Il codice sbagliato non è stato dimenticato; come un giorno andato storto a scuola o un pessimo voto, fa parte della storia, come si scopre da `git log`:

```
$ git log --oneline CowTask.java
0b6b051 That was a bad idea
44b39f3 Deleted variable declaration
e4ee198 Added zombie warning
aa58dfa First Commit
```

Se davvero volessi, potresti anche tornare alla versione errata del codice al commit `44b39f3` o `e4ee198`. Fa parte della bellezza di Git: ricorda tutto, il bello e il brutto, e puoi sempre tornarci nel tempo. Puoi usare la stessa procedura per più file coinvolti nello stesso commit: basta specificarli tutti invece che uno solo. Puoi anche farlo per l’intero progetto; basta non specificare alcun file per `git checkout`, e questo lavorerà sul repository dell’intero progetto.

Visitare più realtà

Poter tornare a un commit precedente è come fare un viaggio nel tempo. Puoi sempre rivisitare il tuo passato. Ma perché fermarsi a un solo passato?

Git permette di gestire delle "diramazioni" (*branch*). Non sono esattamente come i rami di un albero, quanto delle biforcazioni nel continuum spazio-tempo: sono realtà alternative, linee temporali che corrono parallele come nelle storie di fantascienza.

Ecco come puoi usarle. Immagina che tutto funzioni e di aver rilasciato il tuo plug-in nel mondo. Vuoi fare qualche esperimento con un paio di funzioni, ma ti serve avere a disposizione anche la versione corrente tuo plug-in nel caso sia necessario fare delle correzioni per i tuoi utenti. Ti servono allora due *timeline*, cioè due linee temporali: una in cui risiede il plug-in così com'è, magari con un paio di correzioni, e un'altra in cui il plug-in cresce e acquisisce nuove funzionalità. Potresti avere anche una terza timeline in cui implementare tutte le caratteristiche in modo completamente diverso.

Nessun problema!

Puoi creare facilmente una nuova timeline usando il comando `branch` di Git. Di per sé, questo comando elenca tutte le diramazioni attuali nel progetto:

```
$ git branch
* master
```

Di default c'è solo una diramazione, quella principale chiamata `master`, e tu ti trovi in quella timeline. Dividiamo l'universo in due e creiamo una realtà alternativa! È facile (non digitare ancora la prossima riga, ci arriveremo tra un attimo):

```
$ git branch cow-plane
```

Hai creato una nuova timeline, ma non ci sei ancora entrato. Per ora sei ancora in `master`. (Digita `git branch` e vedrai che l'asterisco `*` è ancora vicino a `master`.) Per passare nell'altra timeline, digita

```
$ git checkout cow-plane
```

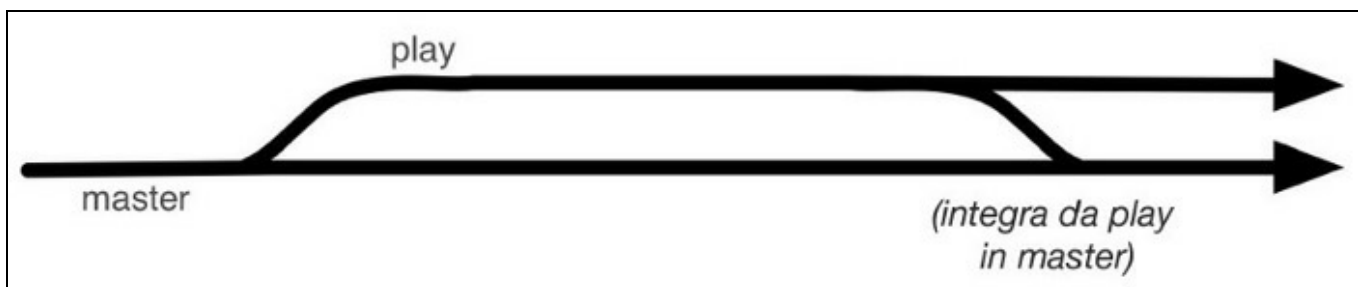
Ebbene sì, è ancora il caro vecchio `git checkout`, che ti ha appena magicamente trasportato in una realtà diversa. Niente di quanto farai qui influenzerà la timeline `master`. Potrai modificare i file, eliminarli, aggiungerli, applicare delle modifiche, e

ogni cosa resterà nell'universo `cow-plane`. Puoi tornare a `master` quando vuoi:

```
$ git checkout master
```

Ora il mondo apparirà com'era nella linea del tempo `master`, con nessuna delle modifiche che hai apportato in `cow-plane`. Correggi i problemi e rilascia questa versione del plug-in, senza i ritocchi che fai man mano alla timeline `cow-plane`.

Prova da solo

In `CowShooter`, creeremo una nuova diramazione chiamata `play`, faremo qualche modifica sia nella diramazione `master` sia in `play` e integreremo le modifiche di `play` in `master`. Ecco lo schema.

La timeline `master` lavora tranquilla, facendosi i fatti suoi. Creiamo una nuova diramazione per giochicchiare e chiamiamola `play`:

```
$ cd ~/Desktop/code/CowShooter
$ git branch play
```

Passiamo poi a quella diramazione usando `git checkout`:

```
$ git checkout play
Switched to branch 'play'
```

Se hai dei dubbi su dove ti trovi, digita `git branch` senza argomenti, e capirai quali sono le diramazioni esistenti; quella in cui ti trovi è contrasssegnata da un asterisco (`*`):

```
$ git branch
  master
* play
```

Ora ci sono due diramazioni, e noi siamo in `play`. Adesso come adesso il contenuto delle due diramazioni è identico. Provvediamo a modificarlo.

Cambiamo `CowShooter.java`. Verso il basso dell'istruzione `if` (che controlla se il giocatore è in possesso del cuoio), c'è una chiamata alla nostra funzione helper, `fling`:

```
...
Canary.getServer().addSynchronousTask(new CowTask(victim));

fling(player, victim, 3);
victim.setFireTicks(600);
...
```

Realizziamo la nostra versione. Crea una nuova funzione in questo plug-in e chiamala `myFling`. Copia la versione da `EZPlugin` nel corpo della funzione, che avrà questo aspetto:

```
public static void myFling(LivingBase player,
                             LivingBase entity, double factor) {
double pitch = (player.getPitch() + 90.0F) * Math.PI / 180.0D;
double rot = (player.getRotation() + 90.0F) * Math.PI / 180.0D;
double x = Math.sin(pitch) * Math.cos(rot);
double z = Math.sin(pitch) * Math.sin(rot);
double y = Math.cos(pitch);

entity.moveEntity(x * factor, y + 0.5, z * factor);
}
```

Ora cambia lo `0.5` nell'ultima riga con un altro numero, per esempio `1.5`. Infine, cambia la chiamata a `myFling` invece che a `fling`:

```
...
Canary.getServer().addSynchronousTask(new CowTask(victim));

myFling(player, victim, 3);
victim.setFireTicks(600);
...
```

Esegui `build.sh` per assicurarti che funzioni ancora e conferma le modifiche:

```
$ git commit -a -m 'Moved vector calculation'
[play 411208c] Moved vector calculation
 1 file changed, 9 insertions(+), 1 deletion(-)
```

Ora che tutto questo è al sicuro in un'istantanea nella diramazione, `play`, torniamo indietro e riprendiamo l'originale:

```
$ git checkout master
Switched to branch 'master'
```

Adesso la versione di `CowShooter.java` su disco è la versione nella realtà `master`. Attenzione: Git ha cambiato il file di testo su disco. La versione nel buffer del tuo editor potrebbe essere ancora quella vecchia. La maggior parte degli editor è abbastanza intelligente da capire quando un file è stato modificato, ma il modo in cui questo viene gestito dipende dall'editor.

Torniamo in `master` e modifichiamo lo spara mucche in modo che diventi uno spara Creeper.

In `CowShooter.java` e in `CowTask.java`, aggiungi l'istruzione di importazione per `Creeper`

in fondo all’elenco delle importazioni:

```
import net.canarymod.api.entity.living.monster.Creeper; // Aggiungi questa riga
```

In `CowShooter.java`, cambia la riga che serve a creare un `Creeper` invece di una `Cow`:

```
Creeper victim = (Creeper)spawnEntityLiving(loc, EntityType.CREEPER);
```

Molto bene! Ora, in `CowTask.java`, modifica la variabile in cima:

```
private Cow cow; // Cancella questa riga
private Creeper creeper; // Aggiungi questa riga
```

Adesso cambia tutti i riferimenti da `cow` a `creeper` perché coincidano (ce ne sono molti).

Al termine, esegui un commit, per sicurezza:

```
$ git commit -a -m ‘Changed to shoot creepers’
[master ea618af] Changed to shoot creepers
 2 files changed, 13 insertions(+), 13 deletions(-)
```

Quando lavoriamo in `master`, abbiamo la fantastica possibilità di importare le modifiche apportate nella diramazione `play`. Per farlo, usiamo `git merge`:

```
$ git merge play -m ‘Bringing over vector calc’
Auto-merging src/cowshooter/CowShooter.java
Merge made by the ‘recursive’ strategy.
 src/cowshooter/CowShooter.java | 8 ++++++++
1 file changed, 8 insertions(+)
```

Se guardiamo il file `CowShooter.java`, vedremo che le modifiche nella diramazione `play` sono state incorporate in `master`, che ora contiene il passaggio alla funzione `fling` e le modifiche da `Cow` a `Creeper`.

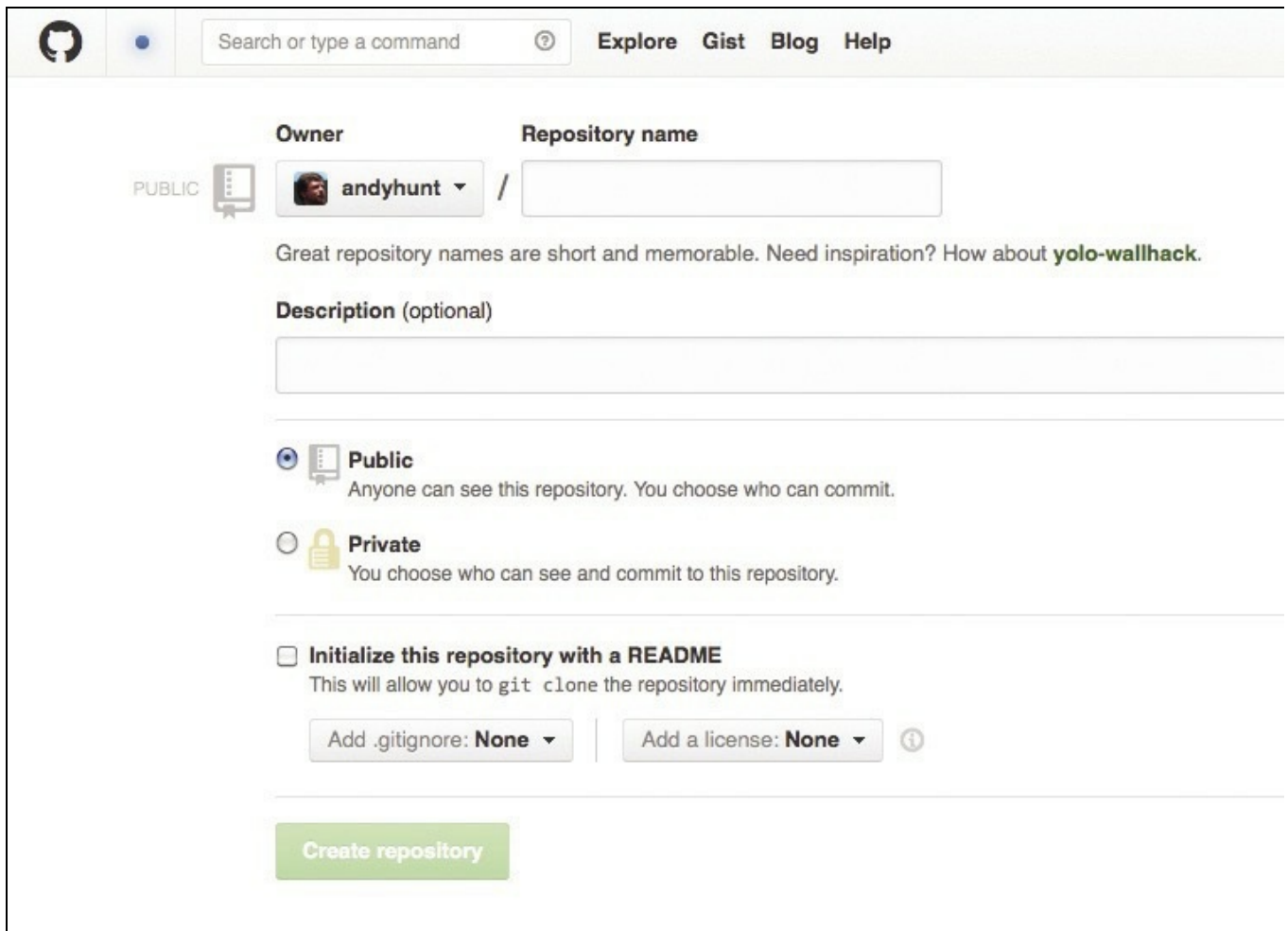
Il fatto che le modifiche siano state integrate non vuol dire che la diramazione `play` sia sparita. È sempre a disposizione e puoi continuare a lavorare con essa quando vuoi. Una volta che hai davvero finito tutto e decidi di non volerla più vedere, ma solo allora, puoi eliminarla:

```
$ git branch -d play
Deleted branch play (was f3e6538).
```

Senza tante cerimonie, verrà rimossa dalla realtà, come mostra `git branch`:

```
$ git branch
* master
```

Backup sul cloud

Questo argomento è molto avanzato. Sentiti libero di saltarlo, all'inizio, ma prima o poi tornaci: dopotutto, gli hard disk si danneggiano, i portatili si rompono e i supporti di salvataggio si perdono…

Git conserva una copia locale di tutti i tuoi file e di tutte le tue modifiche nel suo repository locale, ma puoi impostarlo perché mantenga una copia del tuo repository anche nel cloud.

Puoi decidere quando inviare le modifiche al cloud eseguendo il comando `git push`. Dopo un push, il repository remoto sul cloud avrà lo stesso identico contenuto e le stesse modifiche presenti localmente.

Per avere questa possibilità, ti serve un account su un host Git.

Il più popolare è GitHub, anche se molti preferiscono Bitbucket (http://github.com, http://bitbucket.com). GitHub è lo standard per il codice che intendi condividere con altri o rendere disponibile come open source, mentre Bitbucket è più adatto ai progetti privati che non vuoi condividere con il mondo.

Entrambi offrono interfacce web molto semplici e immediate. Per esempio, una volta che hai creato un account su GitHub, puoi fare clic sul pulsante per creare un nuovo repository, e otterrai una finestra simile alla seguente.

Scegli un nome per il tuo progetto e seleziona *Public* or *Private*. Su GitHub, i repository pubblici sono gratuiti, mentre quelli privati hanno un costo. Bitbucket, invece, offre repository privati illimitati e gratuiti. Non inizializzare il repository, visto che ne hai già uno localmente. Fai semplicemente clic sul pulsante *Create Repository*.

Procediamo. Innanzitutto copia e salva l'URL per il tuo repository in cima. In questo esempio l'URL sarà `https://github.com/andyhunt/minecraft.git`.

Poiché hai già un repository locale, segui le istruzioni in basso per *Push an existing repository from the command line*. Il mio appare così:

```
$ git remote add origin https://github.com/andyhunt/minecraft.git
$ git push -u origin master
```

Il primo comando imposta un punto remoto chiamato `origin` e lo associa all'URL dato. Il push con l'opzione `-u` imposta poi la diramazione `master` in modo che sia tracciata su `origin`. Ogni volta raggiungi un buon punto di interruzione, potrai eseguire `git push` e tutto il codice, e le modifiche verranno salvate al sicuro nel cloud.

Puoi ottenere (tu o chiunque altro, se si tratta di un repository pubblico) una copia del codice e della cronologia delle modifiche usando `git clone` per clonare una copia del repository in una cartella vuota:

```
$ mkdir newcopy
$ cd newcopy
```

```
$ git clone https://github.com/andyhunt/minecraft.git .
```

UN CODICE “AMICHEVOLE”

Quando scrivi del codice che andrà condiviso con altri – cioè, potenzialmente, con tutto il mondo – devi fare un piccolo sforzo in più per fare in modo che sia “amichevole”. Ecco alcuni suggerimenti

- La pulizia conta. Inserisci le parentesi dove necessario. Non cercare di far stare tutto su un’unica riga e usa rientri omogenei. Per i lettori umani, il formato del codice può essere tanto importante quanto il codice stesso.
- Sii breve. Non creare righe o funzioni troppo lunghe.
- Commenta le funzioni. Spiega *perché* una funzione è lì in quel punto e per *cosa* dev’essere usata. Non commentare *come* lo fa; il codice serve a questo.
- Rispetta le convenzioni di denominazione. Java preferisce identificatori scritti in camel-case (come in `minuscoloMaiuscolo`), con le lettere maiuscole in mezzo alla parola. Altri linguaggi possono usare nomi con caratteri di sottolineatura (_) o altre regole. Chiama le variabili in modo tale che i lettori possano capire facilmente qual è il loro scopo. Spesso nei contatori dei cicli vengono usate variabili costituite da una sola lettera (come `i`), ma questa soluzione non è molto descrittiva. Ancora peggio è usare `i` (che solitamente è un indice) come una stringa o una `Location`! Adeguati alle convenzioni.

Condividere il codice

Distribuire il codice tramite GitHub è un ottimo modo per condividerlo con il mondo. Ti basta fornire l'URL GitHub del tuo progetto (in questo esempio `https://github.com/andyhunt/minecraft.git`), dopodiché tutti potranno utilizzare `git clone` per ottenere una copia del tuo codice, e potranno fare le loro modifiche, compilarlo, installarlo e così via.

Uno dei vantaggi di rilasciare il codice al mondo è che gli altri programmatori possono dare il loro contributo aggiungendo funzionalità e correggendo i bachi.

Tanto GitHub quando Bitbucket hanno valide interfacce web che ti facilitano il lavoro.

Immagina che uno dei tuoi fan abbia del codice per una nuova funzione che vorrebbe includere nel tuo plug-in. Creerà una copia del proprio repository ed effettuerà le sue modifiche in una diramazione apposita nel suo repository. In seguito, grazie alla magia di GitHub (o di Bitbucket), ti invierà i cambiamenti contenuti in quella diramazione sotto forma di *pull request*. Ti chiederà cioè di poter inserire la sua diramazione nel tuo repository e di usarla.

Attraverso l'interfaccia web, puoi rispondere alle richieste con domande e commenti, e discutere le modifiche. Quando deciderai di incorporarle, fai clic sul pulsante *Merge Pull Request*. Se il nuovo codice si integra senza conflitti, sei a posto. Altrimenti fai clic sul pulsante *Command Line* e segui le istruzioni che appaiono a video.

Abbiamo appena sfiorato la superficie di Git, che è uno strumento davvero potente e a volte molto complesso. Se vuoi approfondire l'argomento, leggi qualche libro sul tema, come *Pragmatic Version Control Using Git* e *Pragmatic Guide to Git*, entrambi di Travis Swicegood ed editi da the Pragmatic Bookshelf. Ma probabilmente qui c'è tutto quello che ti serve per partire.

Per continuare

Git è uno strumento potente non solo perché ti permette di annullare modifiche importanti nel progetto, ma anche perché rafforza il tuo senso di sicurezza. Non devi temere di mettere tutto in disordine. Se accade, Git ti consente di tornare indietro fino al punto in cui tutto funzionava ancora a dovere.

Nel caso delle diramazioni, puoi addirittura provare due approcci diversi nel codice: non sei sicuro se utilizzare un array o un hash? Non hai la certezza che un listener debba trovarsi in una classe distinta? Le diramazioni permettono di fare un po' di esperimenti senza troppi danni.

Conquistata questa libertà, sei pronto per il passo finale.

`git config --global user.email` *e-mail*	Imposta il tuo indirizzo di posta elettronica.
`git config --global user.name` *"Nome"*	Imposta il tuo nome utente.
`git init`	Inizializza un repository Git nella cartella corrente.
`git add` *file*	Aggiunge i file (o le cartelle) che vuoi vengano tracciati da Git.
`git commit -a -m '`*Messaggio di commit*`'`	Carica le modifiche (tutte in `-a`).
`git log --oneline` *file*	Mostra i messaggi di commit per uno o più file.
`git status`	Mostra quali file sono stati modificati, i file che hai creato ma di cui Git non ha conoscenza (non tracciati) e così via.
`git diff` *file*	Mostra le differenze tra le versioni.
`git checkout` *file*	Rimuove le modifiche locali per i file (da usare con attenzione).
`git checkout` *nome*	Passa alla diramazione con nome.
`git branch`	Elenca tutte le diramazioni.
`git branch` *nome*	Passa alla diramazione con nome
`git remote add` *URL di origine*	Aggiunge un repository remoto come “origine”.
`git push -u origin master`	Copia le modifiche sulla diramazione `master` nel repository `origin`.

Per concludere il nostro viaggio, nel prossimo capitolo vedremo come progettare e costruire un plug-in personale da zero.

La tua toolbox

92%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.
- Usare gli oggetti Java.
- Usare le istruzioni `import` per i package Java.
- Usare `new` per creare gli oggetti.
- Aggiungere un nuovo comando a un plug-in.
- Lavorare con gli oggetti `Location`.
- Trovare blocchi ed entità.
- Usare le variabili locali.
- Usare le variabili globali a livello di classe.
- Usare gli `ArrayList`.
- Usare le `HashMap`.
- Usare `private` e `public` per controllare la visibilità.
- Modificare i blocchi di Minecraft.
- Modificare e generare le entità.
- Monitorare gli eventi di gioco e rispondere.
- Gestire i permessi dei plug-in.
- Creare una classe separata.
- Programmare un'attività che deve essere svolta nel futuro.

- Programmare un’attività in modo che venga svolta in continuazione.
- Salvare e caricare i dati di configurazione.
- Costruire un codice complesso partendo da funzioni più semplici.
- Salvare e caricare i dati di gioco.
- Usare gli oggetti `DataAccess` per lavorare con il database.
- Catturare e sollevare eccezioni in Java.
- Usare Git per tenere traccia delle modifiche al codice.
- Tornare a versioni precedenti del codice (come se si premesse un pulsante *Annulla*).
- Mantenere più versioni del codice contemporaneamente.
- Eseguire un backup del codice sul cloud.

Progettare il plug-in

In tutti i plug-in che hai visto finora, ti ho guidato per mano mostrandoti cosa fare. In questo capitolo ti darò invece alcuni suggerimenti per aiutarti a sviluppare da solo i tuoi plug-in da zero.

Percorreremo insieme i passaggi per creare un nuovo plug-in e vedremo come lavora.

I passi saranno questi (più o meno).

1. Fatti un'idea: "Voglio un plug-in che___________".
2. Raccogli i componenti.
3. Organizzali.
4. Prova ogni singola parte.
5. Riunisci il tutto.
6. Goditela!

Una cosa importante: anche se ho disposto questi passi in ordine e se in ordine li seguiremo, nel mondo reale di solito non succede così.

Raramente la creatività e l'inventiva seguono un percorso diretto e lineare dall'idea all'esecuzione. Scoprirai presto che qualcosa non funziona come dovrebbe, che il codice è tutto sbagliato o magari che è giusto ma non fa quello per cui l'avevi pensato. È normale e tipico della programmazione.

Potresti dover buttare via qualcosa di quanto hai creato e ricominciare, a volte anche l'intero progetto. Capita anche allo Stato di sprecare soldi su progetti sbagliati, quindi può succedere a anche a te.

I programmatori professionisti affrontano il problema procedendo per piccoli passi: fai una cosa alla volta, accertati che funzioni e poi passa a quella successiva. Ogni volta che ti rendi conto di aver commesso un errore o aver frainteso qualcosa, torna indietro e correggi là dove serve. Non rimandare convinto che ti ricorderai di farlo più avanti: non lo farai. Fidati.

Buttiamoci nella mischia, allora, e progettiamo un nuovo plug-in. Qui ti

proporrò le tappe per un plug-in mio, e tu mi seguirai in parallelo progettando il tuo. Il tuo plug-in potrebbe fare qualcosa di diverso, ed è probabile che ci voglia un po' di tempo per arrivare in fondo: questo non è il tipo di capitolo che fai fuori in una sera…

Fatti un’idea

Prima di iniziare, devi riflettere su quello che vuoi scrivere. Potrebbe non essere ancora un’idea eccezionale o perfetta, ma andrà bene lo stesso. Questo è software, dopotutto, e ti è concesso qualche ripensamento. Da qualche parte bisogna comunque partire.

Il mio progetto è quello di generare casualmente delle “mucche-Creeper”, cioè delle mucche che saltano fuori e provano ad attaccare i giocatori, esplodendo quando saltano loro addosso. Potrebbe essere divertente.

Ora come ora so solo questo, e non ho la minima idea di come iniziare.

Prova da solo

Qual è la tua idea? Annotala da qualche parte. Se sei a corto di ispirazione, dai un’occhiata ai plug-in scritti da altri, oppure sfoglia la documentazione di Canary per trovare qualche spunto.

Raccogli i componenti

Trovata un'idea, devi "raccogliere i componenti". Se intendi fabbricare qualcosa, devi radunare tutti i materiali (colla, legno, carta, plutonio) e gli strumenti (forbici, martello, saldatore e così via). Lo stesso vale per noi: quali materiali ci servono?

Per capirlo, dobbiamo prima determinare che tipo di dati serviranno al plug-in, come tenerne traccia e cosa fa girare il plug-in: è un comando di esecuzione, un evento o un timer? Ecco alcune domande a cui rispondere.

- Di cosa vuoi tenere traccia?
- Per quanto a lungo vuoi tenerne traccia? (Durante un comando, mentre il server è in esecuzione, tra i vari riavvii del server su disco…)
- Qual è il *trigger* che lo innesca? (Un comando immesso dall'utente, un evento di gioco, uno stato interno, un timer, una combinazione…)
- Su quali parti del gioco dovrai influire (blocchi, giocatori, pozioni, inventario…) e quali funzioni Canary o Java dovrai utilizzare?
- Cosa potrebbe andare storto?

Potresti non avere ancora tutte le risposte, ma questo non deve fermarti. Partiamo seguendo la mia idea delle mucche-Creeper.

Di cosa vuoi tenere traccia? Qui probabilmente dovremo tenere traccia delle mucche che generiamo, dei giocatori bersaglio che ogni mucca sta puntando e cercando di attaccare, o magari di altro che capiremo solo procedendo. Nient'altro? Oh, sì, vogliamo creare delle posizioni casuali per le mucche: ti ricordi quale funzione usare per ottenere numeri casuali in Java?

Per quanto a lungo vuoi tenerne traccia? Poiché queste sono mucche d'attacco generate casualmente, non ci serve sapere che fine fanno dopo che il server è stato fermato: al riavvio, ne creeremo di nuove. Quello di cui abbiamo bisogno, invece, è seguirle mentre il server è in esecuzione, quindi ci serve almeno una lista o un hash statico delle mucche che abbiamo generato. Se ti serve un ripasso, rileggi il Capitolo 7.

Quali sono i trigger? Sarebbe piuttosto stupido dover digitare un comando per essere attaccati dalle mucche, quindi non utilizzeremo una funzione `@Command`.

Piuttosto, ci serve un evento che avvii la creazione delle mucche, e magari un timer per lanciare le mucche all'attacco e infine per farle esplodere. Dobbiamo trovare un evento adatto da ascoltare. Per i dettagli, rileggi i Capitoli 9 e 11.

Su quali parti del gioco dovrai influire? Dobbiamo spostare qua e là le mucche assassine e poi farle esplodere, come abbiamo imparato a fare nei plug-in precedenti, e dobbiamo distruggere anche il giocatore che viene attaccato. Possiamo far avvenire l'esplosione quando la mucca è vicina al giocatore o dobbiamo uccidere il giocatore esplicitamente (impostando la sua salute a zero, dandogli fuoco o altro del genere?) Dovremo fare qualche esperimento e vedere come funziona.

Cosa potrebbe andare storto? Questa è una domanda da porsi continuamente quando si crea un plug-in. Ora come ora è probabile che le mucche vaghino confuse durante l'attacco o che restino bloccate da qualche parte, oppure che non ci sia un giocatore lì vicino da attaccare. Dovremo provvedere. Una buona idea è quella di limitare il numero di mucche che verrà creato, così da evitare una pioggia di mucche…

Non è molto, ma è un inizio. Ecco cosa abbiamo raccolto finora.

- Una lista o un hash statico delle mucche che abbiamo generato.
- La funzione Java che crea un numero casuale (ricordi qual è questa funzione?)
- Un evento che inneschi la creazione delle mucche.
- L'idea che ogni mucca debba avere un timer, così che possa trovare i giocatori e attaccarli.
- L'idea che la mucca non debba restare bloccata sul terreno (cosa fare se non ci sono giocatori da attaccare lì intorno?).
- La decisione di dover porre un limite al numero delle mucche generate.

Prova da solo

Ripercorri queste domande con in mente il plug-in che vuoi costruire, e fai un elenco dei componenti che ti servono.

Includi anche quelle cose che non sei certo di utilizzare: non pagherai alcuna penalità se poi non le userai.

Organizza i componenti

Prima di creare il plug-in e iniziare a scrivere il codice, pensiamo a dove deve finire tutto quello che abbiamo raccolto. In altre parole, rientrerà tutto in una funzione o in una classe? E se ci sono più funzioni o classi, cosa andrà dove? E se ci sono altre classi, cosa devono sapere di noi o di loro stesse? Se hai dimenticato i dettagli delle funzioni e degli oggetti, torna ai Capitoli 4 e 5 per un ripasso.

Le domande da porsi sono queste.

- Cosa va dove? Quali funzioni e classi ti servono?
- Perché ti serve quella funzione (o quella classe)? Cosa dovrà fare?
- Chi altri, nel caso, deve sapere di questa funzione o classe?

Un modo di procedere, soprattutto con i plug-in più complessi, è quello di utilizzare le schede CRC (CRC sta per *classe-responsabilità-collaboratori*), inventate dai nostri amici Ward Cunningham e Kent Beck per organizzare i progetti software. Dividi ogni scheda in tre parti: il nome dell’oggetto in alto, un elenco di cose di cui è l’oggetto responsabile a sinistra e le altre classi necessarie per lavorare a destra.

Nel caso della classe principale del nostro plug-in, la scheda appare così:

CreeperCow	
Mantenere una lista di mucche	Cow
Creare nuove mucche e attivarle	Eventi Canary
Eliminare le mucche morte	CreeperCowTimer

C’è poi il `CreeperCowTimer`, che gestirà la mucca e si occuperà di tutto quanto deve fare una mucca che va all’attacco:

CreeperCowTimer	
Attaccare i giocatori	CreeperCow Attività del server

Nota che non siamo ancora entrati nel dettaglio delle singole funzioni: prima è necessario farsi un'idea precisa del perché una determinata classe deve esistere. Di cosa è responsabile?

Per ogni responsabilità, prova a capire quello che serve. Per non restare troppo legato alla sintassi Java (o a quella di altri linguaggi di programmazione), esprimi il concetto in italiano. Racconta la storia di *quello* che vuoi fare, non ancora di *come* vuoi farlo.

Ogni passo diventerà una funzione, o *più funzioni* se decidi di suddividere il passo in un paio di funzioni e/o classi più semplici.

1. Creiamo la lista di mucche.
2. Generiamo mucche-Creeper e aggiungiamole alla lista.
3. Impostiamo un timer per ogni mucca-Creeper.
4. Ascoltiamo gli eventi per creare ed eliminare le mucche.

Questa parte è stata semplice. Ma cosa deve fare una mucca-Creeper? Abbiamo detto che il suo compito è quello di attaccare i giocatori. Cioè? Vediamo.

1. Trova il giocatore più vicino player (potrebbe non essercene neanche uno).
2. Salta verso il giocatore più vicino (che non morirà nell'impatto).
3. Se colpisce il giocatore più vicino, esplode.

Nel caso del plug-in `CreeperCow`, prendiamo la lista dei componenti raccolti e dei vari passi, creiamo alcune funzioni e vediamo il codice necessario.

`CreeperCow` (la classe principale del plug-in):

- `cowList`: una lista statica di mucche che abbiamo creato;
- `Math.random()`: ci serve un numero casuale; la documentazione Java dice che

restituirà un double tra 0 e 1 (`http://bit.ly/1v9VGRz`);

- `spawnCows()`: crea un numero casuale di mucche che non deve superare un limite prestabilito. Creeremo un `CreeperCowTimer` per gestire ogni mucca e lo attiveremo;
- `eventListener()`: un listener per chiamare `spawnCows()`.

`CreeperCowTimer` (uno per mucca):

- `findClosestPlayer()`: potrebbe non esserci un giocatore vicino;
- `jump(Location loc)`: fare un *jump* significa spiccare un balzo per aria e non morire nell'impatto;
- `explode()`: se colpiamo un giocatore (o se siamo abbastanza vicini), la mucca esplode e il plug-in la elimina.

Resta ancora molto da fare, ma ce n'è abbastanza per iniziare ad assemblare del codice. Ci aiuterà a rispondere anche alle domande rimanenti, e potrebbe anche sollevarne delle altre. Procediamo

Prova da solo

Prendi qualche scheda CRC o dei fogli di carta e annota la tua classe principale e le sue responsabilità e le parti di Canary che possono servirti. Se ti occorre qualche classe in più che hai visto qui (per esempio qualcosa che avvia un'attività) o qualcos'altro di cui ti vuoi tenere traccia, crea delle schede anche per quello.

Per ogni responsabilità, scrivi un elenco di passi che si possono leggere come una storia. Poi percorri l'elenco dei componenti e prova a creare una lista di funzioni e variabili, ponendo le stesse domande che abbiamo posto qui.

Una volta assemblata la lista, è il momento di iniziare a scrivere del codice, anche se non tutto, come vedremo nel prossimo paragrafo.

Testa ogni singola parte

Cominceremo adesso a scrivere il codice, ma per quanto sia emozionante, dovremo andare con calma. Che tu segua il mio plug-in o che lavori sul tuo, ricorda di procedere un passo alla volta: non avere fretta.

Iniziamo creando una nuova cartella e i soliti file che servono per ogni plug-in. Per partire useremo ancora una volta lo script `mkplugin.sh`. Ricorda che creerai il plug-in sul desktop; nel mio caso la cartella è *Desktop/code*:

```
~/Desktop$ ./mkplugin.sh CreeperCow
~/Desktop$ cd CreeperCow
~/Desktop/CreeperCow$ ls -a
.            .gitignore     Manifest.txt build.sh    src
..            Canary.inf    bin                      dist
```

Usiamo fin dall'inizio anche Git per tenere traccia del nostro lavoro (per un ripasso su Git rileggi il Capitolo 12).

Lo script `mkplugin.sh` ha creato per noi un `.gitignore` che ignorerà le cartelle *bin/* e *dist/*. Aggiungiamo la nostra prima coppia di file e carichiamola come base di partenza:

```
~/Desktop/CreeperCow$ git init
Initialized empty Git repository in /Users/andy/Desktop/CreeperCow/.git/
~/Desktop/CreeperCow$ git add .gitignore build.sh Canary.inf Manifest.txt src
~/Desktop/CreeperCow$ git commit -a -m First
[master (root-commit) 2e483f1] First
 5 files changed, 95 insertions(+)
 create mode 100644 .gitignore
 create mode 100755 build.sh
 create mode 100644 Canary.inf
 create mode 100644 Manifest.txt
 create mode 100644 src/creepercow/CreeperCow.java
```

(Se sei collegato a un repository remoto, puoi anche eseguire un `git push`.) E ora il codice.

Sappiamo di dover creare una `cowList` statica e un metodo `spawnCows()`, e dovremo anche ascoltare alcuni eventi, quindi lavoriamo in `CreeperCow.java`. Ecco la parte interessante (ho omesso le altre istruzioni di importazione e la solita roba):

```
import net.canarymod.plugin.PluginListener;
import net.canarymod.api.entity.living.animal.Cow;
import java.util.ArrayList;
import com.pragprog.ahmine.ez.EZPlugin;

public class CreeperCow extends EZPlugin implements PluginListener {

  private static ArrayList<Cow> cowList = new ArrayList<Cow>();

  public void spawnCows() {

}
```

Abbiamo reso il plug-in principale un `PluginListener` e abbiamo aggiunto una variabile `cowList` all'inizio della funzione `spawnCows()`. Prima di proseguire, testiamo questa parte per vedere se si compila senza errori e poi eseguiamo la funzione (vuota) `spawnCows`.

Piccoli passi, ricorda. Non abbiamo ancora un listener (in realtà non sappiamo nemmeno quale evento ascolteremo), quindi usiamo un trucchetto.

Aggiungeremo un comando a `@Command` esclusivamente a nostro uso e consumo – non è per gli utenti – da impiegare per testare quanto realizziamo man mano.

Partiamo subito da questo, anche prima di aver aggiunto codice a `spawnCows`:

```
@Command(aliases = { "testspawncows" },
    description = "Test cow spawning",
    permissions = { "" },
    toolTip = "/testspawncows")
public void testSpawnCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    Location loc = me.getLocation();
   spawnCows();
  }
}
```

(Ci serve anche `import net.canarymod.api.world.position.Location;`.)

Adesso possiamo entrare nel gioco ed eseguire il comando `/testSpawnCows` per veder spuntare le mucche e assicurarci che il trucchetto funzioni. Non abbiamo ancora generato delle mucche, ma proviamo a farlo adesso, prima di inserire altro codice, per essere certi che tutto si compili bene, usando `build.sh`:

```
~/Desktop/CreeperCow$ build.sh
Compiling with javac...
Creating jar file...
Deploying jar to /Users/andy/Desktop/server/plugins...
Completed Successfully.
```

È un buon metodo per controllare di aver importato quanto necessario senza aver commesso refusi. Funziona, quindi facciamo un `git commit`.

Prova da solo

Ora tocca a te: crea una nuova cartella per il plug-in con `mkplugin.sh`, imposta un repository Git e aggiungi le funzioni e i dati che pensi possano servirti. Se hai bisogno di una rinfrescata su certi argomenti, riprendi le parti precedenti del libro.

Aggiungi i comandi per il test a `@Command`, oltre ad aggiungere altri comandi che

hai già. Piccoli passi, sempre. Vuoi che le funzioni più piccole si eseguano in modo indipendente, e che lo facciano correttamente prima di continuare. Non riempirle ancora: per ora inizia con funzioni vuote come hai visto qui.

Prima di continuare, non dimenticare di eseguire un `git commit` per salvare tutto.

Entrare nei dettagli: la funzione spawnCows()

Ora che sappiamo come testarla, entriamo nel cuore della funzione `spawnCows()`. Abbiamo già visto che dovremo fare quanto segue.

1. Creare la lista di mucche.
2. Generare mucche-Creeper e aggiungerle alla lista.
3. Ascoltare gli eventi per creare ed eliminare le mucche.

Abbiamo già creato la lista statica di mucche, quindi passiamo alla loro generazione.

Dovendo creare delle mucche casualmente, nasce una domanda: dove metterle? Vanno distribuite in tutto il gioco? Vicino alla nostra posizione? Propria sopra alla testa del nostro amico? Sono tutte possibilità interessanti.

Una delle strategie migliori della programmazione consiste nel ritardare le decisioni. In molti casi non conosci subito la risposta “giusta”, e potresti non conoscerla per un bel po’, se non addirittura mai.

È proprio a questo che servono i parametri di una funzione: non dobbiamo decidere tutti i dettagli adesso. Invece di implementare i particolari, passiamoli come parametro. La decisione è rimandata, e la funzione può comunque essere utilizzata con tutti i tipi di dettaglio.

Dovremo modificare la dichiarazione temporanea che avevamo per `spawnCows()` e passare qualcosa per indicare dove vogliamo collocare le mucche.

Partiamo da una `Location`, cioè da un quadrato di blocchi. La posizione sarà costituita da un angolo di un grande quadrato che verrà esteso di un certo numero di blocchi nelle direzioni x e z. Cambiamo allora la funzione come segue:

```
public void spawnCows(Location start, int size) {
```

Più avanti decideremo cosa passare. Per gli scopi del test possiamo passare la posizione del `Player` e un numero a scelta.

C’è un piccolo ostacolo: una funzione passa gli argomenti a un comando dell’utente sotto forma di un array di oggetti `String`. Noi però non vogliamo passare a `testSpawnCows()` una `string`, ma un `int`. Come ottenere questa conversione?

Basta fare una ricerca in Google per “conversione di stringhe Java in int” per trovare la risposta:

```
int foo = Integer.parseInt("1234");
```

Ora possiamo rifinire il comando per chiamare `spawnCows` dal nostro comando di test:

```
@Command(aliases = { "testspawncows" },
    description = "Test cow spawning",
    permissions = { "" },
    min = 2, // Numero di argomenti
    toolTip = "/testspawncows <number to spawn>")
public void testSpawnCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    Location loc = me.getLocation();
    spawnCows(loc, Integer.parseInt(args[1]));
  }
}
```

Adesso sì che possiamo aggiungere il codice per implementare `spawnCows`, perché abbiamo un modo per controllarlo. I professionisti lavorano così: sviluppano un modo per verificare il codice prima di scrivere il codice vero e proprio. Possiamo farlo anche noi quasi automaticamente, visto che dobbiamo ancora entrare manualmente nel gioco e avere la conferma visiva che stiamo generando delle mucche-Creeper, ma l’idea è quella.

E ora iniziamo a generare un po’ di mucche! Useremo un ciclo `for`, la posizione di partenza e le dimensioni per creare una nuova posizione in cui creare ogni mucca. Quante mucche creeremo? Ops, non ci abbiamo ancora pensato. Aggiungiamolo alla segnatura della funzione:

```
public void spawnCows(Location start, int size, int number) {
```

e passiamolo da `testSpawnCommand()` per un test:

```
spawnCows(loc, Integer.parseInt(args[1]), Integer.parseInt(args[2]));
```

Ora sappiamo come far creare mucche al loop `for` e come aggiungerle alla lista delle `Cow`. Inseriamo infine il codice in `spawnCows`:

```
public void spawnCows(Location target, int size, int count) {
  World world = target.getWorld();
  double x = target.getX();
  double z = target.getZ();
  for (int i=0; i< count; i++) {
    Location loc = new Location(world,
```

```
      x + (Math.random() * size),
      0,
      z + (Math.random() * size),
      0,0
    );
    loc.setY(world.getHighestBlockAt((int)loc.getX(), (int)loc.getZ()) + 2);
    Cow cow = (Cow)spawnEntityLiving(loc, EntityType.COW);
  }
}
```

Moltiplichiamo il numero casuale (0..1) per la dimensione del quadrato, e otterremo un numero compreso tra zero e la dimensione del quadrato. Useremo questo valore per la x e la z della posizione sul quadrato, dopodiché chiediamo al server il blocco più in alto in quel punto e usiamolo come y. Lì è dove creeremo la nuova mucca.

Compiliamo, costruiamo e testiamo.

Nel mio caso, mi collego al gioco e provo il comando `/testspawncows 10 8`, che genera otto mucche in un blocco di 10×10 dalla mia posizione corrente.

Ehi, abbiamo creato delle mucche! Ora come ora non fanno così paura, però. Se ne stanno lì ferme, come fanno sempre le mucche. Invece ci serve che saltino verso l’alto e attacchino.

Ho scritto del codice e l’ho testato; prima di procedere salverò questo stato del mondo con un `git commit`.

Prova da solo

È il tuo turno. Prova a riempire le funzioni come hai appena visto. Non devono fare tutto adesso, basta riempirle con qualcosa per partire (per esempio, le nostre mucche non saltano ancora).

Per ogni funzione che crei, includi un comando di test per provarla all'interno del gioco.

Assicurati che i comandi di test funzionino per tutto quello che hai creato finora, e non dimenticare di salvare i tuoi progressi con Git.

Entrare nei dettagli: CreeperCowTimer

In precedenza abbiamo deciso di inserire in una nuova classe `CreeperCowTimer` il codice per far saltare, attaccare ed esplodere le mucche. Per finire, avvieremo tutto questo da un evento nel plug-in principale `CreeperCow`.

Prima, però, ci serve una funzione che porti le mucche a saltare e ad attaccare. Costruiamola e testiamola.

Ottenuti un bersaglio e la posizione della mucca, creiamo un nuovo `Vector3D` prendendo la differenza nelle coordinate di `Location` nelle direzioni x e z: questa differenza ci fornirà l'entità dello spostamento tra la mucca e il suo bersaglio.

È un numero però troppo grande per una velocità, quindi moltiplichiamolo, per esempio, per 0.075 (il valore che mi è sembrato più adatto dopo aver fatto qualche esperimento).

Impostiamolo come velocità della mucca, che salterà lungo questo `Vector3D` a questa velocità:

```
public class CreeperCowTimer {
  private Cow cow;

  public void jump(Location target) {
    Location cowLoc = cow.getLocation();
    double multFactor = 0.075;
    Vector3D v = new Vector3D(
      (target.getX() - cowLoc.getX()) * multFactor,
      0.8,
      (target.getZ() - cowLoc.getZ()) * multFactor
    );
    cow.moveEntity(v.getX() + (Math.random() * -0.1),
      v.getY(),
      v.getZ() + (Math.random() * -0.1));
  }
}
```

Ho caricato il nuovo file `CreeperCowTimer.java` su Git con `git add` per tenere traccia delle modifiche.

Ora ci occorre un modo per impostare l'oggetto mucca come privato. Più tardi lo collegheremo alla funzione `spawnCows`, ma per ora passeremo semplicemente una mucca che è stata già creata. Per farlo, usiamo la funzione del costruttore:

```
CreeperCowTimer(Cow aCow) {
  cow = aCow;
}
```

A seguire dobbiamo aggiungere al plug-in un comando per il salto per testarlo. Torniamo alla funzione `spawnCows` e aggiungiamo le mucche create all'`ArrayList`. Poi, nel comando per il salto, possiamo percorrere nell'elenco e far saltare ogni mucca verso di noi.

Partiamo con questo:

```
Cow cow = (Cow)spawnEntityLiving(loc, EntityType.COW);
cowList.add(cow);
```

poi aggiungiamo un comando di test:

```
@Command(aliases = { "testjump" },
  description = "Test cow jumping",
  permissions = { "" },
  min = 1, // Numero di argomenti
  toolTip = "/testjump")
public void testJumpCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
      for (Cow c : cowList) {
        c.jump(me.getLocation());
      }
   }
}
```

Accidenti, non funziona.

Non avremmo dovuto tenere una lista di mucche, bensì tenere traccia degli oggetti `CreeperCowTimer`, poiché è il timer che attiva la funzione `jump()` e il cuore di quanto dovremo scrivere, oltre alla mucca stessa.

Se procediamo così, probabilmente dovremo anche modificare l'`ArrayList` in un'`HashMap`, così da poter considerare gli oggetti `CreeperCowTimer` in base alla rispettiva `Cow`, che è proprio quanto otteniamo dagli eventi, dalla generazione delle mucche e così via.

Dobbiamo fare qualche modifica. Fai anche tu un elenco dei cambiamenti necessari e poi torna qui.

Modifiche necessarie

Ecco le modifiche da apportare.

1. Modificare una `import java.util.ArrayList;` in una `import java.util.HashMap;`.
2. Modificare la `cowList` da `public static ArrayList<Cow> cowList = new ArrayList<Cow>();` in un hash: `public static HashMap<Cow, CreeperCowTimer> allCows = new HashMap<Cow, CreeperCowTimer>();`. Ho anche rinominato l'hash `allCows`, così che il nome non contenga il tipo di lista.
3. Modificare `cowList.add(cow);` in `allCows.put(cow, new CreeperCowTimer(cow));` per il `put` in un hash.
4. Modificare il ciclo `for` per iterare lungo l'`HashMap` di modo che il nuovo ciclo appaia così:

```
for (Cow c : allCows.keySet()) {
  CreeperCowTimer superCow = allCows.get(c);
  superCow.jump(me.getLocation());
}
```

Può sembrare frustrante, ma non lo è. Le cose non vanno sempre bene al primo colpo, ma non è un problema. Quando anche girasse tutto subito, non ne trarresti grandi vantaggi. Quello che conta è che tutto funzioni *alla fine*.

Una volta apportate le modifiche, possiamo testare il salto. Avvio il server, mi connetto dal client e provo a creare una singola mucca. Poi uso il comando `testJump` per vedere se la mucca inizia a dirigersi verso di me:

```
/testspawncows 5 1
```

Lo fa! E ora qualche salto:

```
/testjump
/testjump
/testjump
```

Ops, la mucca è morta. È lo stesso inconveniente incontrato nel plug-in

`CowShooter`: dobbiamo impostare la salute della mucca al massimo:

```
cow.setHealth(cow.getMaxHealth());
```

Ma dove farlo? E cos’altro server per completare questo plug-in? Questa versione gira, ma ha un problema. Esegui un `git commit` per salvare lo stato attuale del codice in questo punto prima di compiere altre modifiche importanti.

Assemblare il tutto

Dovremo aggiungere un paio di cose dal nostro elenco di componenti e occuparci di alcuni problemi.

Per prima cosa dovremo evitare che le mucche muoiano, quindi aggiungiamo un gestore di eventi che ascolti `DamageHook` e annulli l'evento se la causa è un `DamageType.FALL`. Attenzione: se cancelliamo questo evento, non ci servirà alzare il livello di salute della mucca.

Quando abbiamo testato la funzione di generazione e il salto, abbiamo usato noi stessi come bersagli. Dobbiamo fare qualcosa di meglio, quindi aggiungiamo una funzione `getClosestPlayer()` per trovare il giocatore più vicino a una mucca.

Come abbiamo già visto, dobbiamo scegliere un evento da usare per generare le mucche-Creeper. Abbiamo a disposizione due modi per farlo. Visto che non vogliamo distribuire mucche per tutto il mondo, sarebbe bello se ci fosse un evento che viene inviato ogni volta che nel server viene caricato un nuovo trancio tettonico del mondo (un *chunk*).

La documentazione di Canary sotto a `net.canarymod.hook.world`, ci svela quello che ci serve: un `ChunkLoadedHook` quando viene caricato un chunk e un `ChunkUnloadHook` quando viene eliminato. Così, ogni volta che viene caricato un nuovo trancio di 16×16 blocchi, otterremo un evento. Attenzione: il trancio ci dice a quale segmento corrisponde all'interno di una griglia di tranci: per ottenere una coordinata del mondo reale, dobbiamo moltiplicare per 16.

Adesso la nostra lista delle cose da fare include queste voci.

- Aggiungere un gestore di eventi `DamageHook` per impedire che la mucca muoia.
- Aggiungere `getClosestPlayer` perché ogni mucca trovi un bersaglio nelle vicinanze.
- Aggiungere `ChunkLoadedHook` per generare le mucche in un'area di 16×16 blocchi.
- Aggiungere il `ChunkUnloadHook` per rimuovere le mucche da quel trancio.
- Impostare un timer di attività in modo che ogni mucca trovi un bersaglio, salti verso di esso ed esploda se necessario.
- Gestire `allCows` in tutte queste funzioni (aggiungerlo alla creazione, eliminarlo

con la morte o annullare il caricamento).

È un bel po' di lavoro. Sono tutte attività che usano funzioni e caratteristiche che abbiamo già impiegato, quindi non ti annoierò di nuovo con le procedure.

Ecco il codice nella sua interezza, da leggere e dal quale copiare, se ti serve.

CreeperCow/src/creepercow/CreeperCow.java

```
package creepercow;

import java.util.HashMap;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.Collection;
import net.canarymod.plugin.Plugin;
import net.canarymod.logger.Logman;
import net.canarymod.Canary;
import net.canarymod.commandsys.*;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.api.entity.Entity;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.world.World;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.position.Vector3D;
import net.canarymod.hook.HookHandler;
import net.canarymod.api.inventory.ItemType;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.api.entity.EntityType;
import net.canarymod.api.entity.living.animal.Cow;
import net.canarymod.api.entity.EntityType;
import net.canarymod.plugin.PluginListener;
import net.canarymod.hook.world.ChunkLoadedHook;
import net.canarymod.hook.world.ChunkCreatedHook;
import net.canarymod.hook.world.ChunkUnloadHook;
import net.canarymod.hook.entity.DamageHook;
import net.canarymod.api.DamageType;
import net.canarymod.api.world.Chunk;
import com.pragprog.ahmine.ez.EZPlugin;

public class CreeperCow extends EZPlugin implements PluginListener {

  private static HashMap<Cow, CreeperCowTimer> allCows =
    new HashMap<Cow, CreeperCowTimer>();

  private static boolean enabled = false;

  private final static int CHUNK_SIZE = 16;
  @Override
  public boolean enable() {
    Canary.hooks().registerListener(this,  this);
    return super.enable(); // Chiama anche la versione genitore della classe.
  }

  public void spawnCows(Location target, int size, int count) {
    World world = target.getWorld();
    double x = target.getX();
    double z = target.getZ();
    for (int i=0; i< count; i++) {
      Location loc = new Location(world,
        x + (Math.random() * size),
```

```
        0,
        z + (Math.random() * size),
        0,0
      );
      loc.setY(world.getHighestBlockAt((int)loc.getX(), (int)loc.getZ()) + 2);
      logger.info("[CreeperCow] spawned cow at " + printLoc(loc));
      Cow cow = (Cow)spawnEntityLiving(loc, EntityType.COW);
      CreeperCowTimer task = new CreeperCowTimer(this, cow);
      Canary.getServer().addSynchronousTask(task);
      allCows.put(cow, task);
    }
  }

  public void cowDied(Cow cow) {
    logger.info("[CreeperCow] cow died.");
    allCows.remove(cow);
  }

  @Command(aliases = { "creepercows" },
    description = "Turn Creeper Cows on and off",
    permissions = { "" },
    min = 2, // Numero di argomenti
    toolTip = "/creepercows on|off")
public void enabledCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    if (args[1].equalsIgnoreCase("on") ||
      args[1].equalsIgnoreCase("yes") ||
      args[1].equalsIgnoreCase("true"))  {
      enabled = true;
      me.chat("Creeper Cows are enabled");
      // Parti da qui con qualcuna ;)
      spawnCows(me.getLocation(), 25, 5);
    } else {
      enabled = false;
      me.chat("Creeper Cows are disabled");
    }
  }
}

@Command(aliases = { "testspawncows" },
    description = "Test cow spawning",
    permissions = { "" },
    min = 3, // Numero di argomenti
    toolTip = "/testspawncows <size of square> <number to spawn>")
public void testSpawnCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    Location loc = me.getLocation();
    spawnCows(loc,
        Integer.parseInt(args[1]),
        Integer.parseInt(args[2]));
  }
}

@Command(aliases = { "testjump" },
    description = "Test cow jumping",
    permissions = { "" },
    min = 1, // Numero di argomenti
    toolTip = "/testjump")
public void testJumpCommand(MessageReceiver caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    for (Cow c : allCows.keySet()) {
      CreeperCowTimer superCow = allCows.get(c);
      superCow.jump(me.getLocation());
```

```
    }
  }
}

@Command(aliases = { "testexplode" },
    description = "Test cow explode",
    permissions = { "" },
    min = 1, // Numero di argomenti
    toolTip = "/testexplode")
public void testExplodeCommand(MessageReceiver  caller, String[] args) {
  if (caller instanceof Player) {
    Player me = (Player)caller;
    List<CreeperCowTimer> list = new ArrayList<CreeperCowTimer>();
    for (Cow c : allCows.keySet()) {
      CreeperCowTimer superCow = allCows.get(c);
      list.add(superCow);
    }
    for (CreeperCowTimer superCow : list) {
      superCow.explode();
    }
  }
}

@HookHandler
public void onChunkLoad(ChunkLoadedHook event) {
  if (enabled) {
    World world = event.getWorld();
    Chunk chunk = event.getChunk();

    if (Math.random() > 0.10) { // Crea solo una mucca su dieci
        return;
    }
    logger.info("[CreeperCow] Spawning");
    // La X e la Z del chunk sono indici;
    // dobbiamo moltiplicare per 16 per ottenere
    // la posizione reale del blocco.
    Location start = new Location(
        chunk.getX() * CHUNK_SIZE,
        0,
        chunk.getZ() * CHUNK_SIZE);
    spawnCows(start, 16, 1);
  }
}

@HookHandler
public void onChunkUnload(ChunkUnloadHook event) {
  Chunk chunk = event.getChunk();
  List<Entity>[] all = chunk.getEntityLists();
  for(int i = 0; i < all.length; i++) {
    for (Entity ent : all[i]) { // Elenco di sottochunk di 16 blocchi
      if (ent instanceof Cow) {
        Cow cow = (Cow) ent;
        if (allCows.containsKey(cow)) {
        allCows.get(cow).removeMe();
        allCows.remove(cow);
        }
      }
    }
  }
}

@HookHandler
public void onEntityDamage(DamageHook event) {
  Entity ent = event.getDefender();
  if (ent instanceof Cow) {
    Cow cow = (Cow) ent;
```

```
    if (event.getDamageSource().getDamagetype()  == DamageType.FALL) {
      if (allCows.containsKey(cow)) {
        event.setCanceled();
      }
    }
  }
}

}
```

CreeperCow/src/creepercow/CreeperCowTimer.java

```
package creepercow;

import java.util.List;
import java.util.ArrayList;
import net.canarymod.Canary;
import net.canarymod.api.entity.EntityType;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.entity.living.animal.Cow;
import net.canarymod.api.world.position.Vector3D;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.tasks.ServerTask;
import com.pragprog.ahmine.ez.EZPlugin;


public class CreeperCowTimer extends ServerTask {
  private Cow cow;
  private CreeperCow plugin;

  CreeperCowTimer(CreeperCow parentPlugin, Cow aCow) {
  super(Canary.getServer(), 0, true); // Ritardo, isContinuous
  cow = aCow;
  plugin = parentPlugin;
}

public Player getClosestPlayer(Location loc) { //Restituisce -1 in caso di errore
List<Player> list = Canary.getServer().getPlayerList(); Player closestPlayer = null;
double minDistance = -1;
for(int i = 0; i < list.size(); i++) {
  Player p = list.get(i);
  Location ploc = p.getLocation();
  if (Math.abs(ploc.getY() - loc.getY()) < 15) {
    double dist = distance(loc, ploc);
    if (dist < minDistance || minDistance == -1) {
      minDistance = dist;
      closestPlayer = p;
    }
  }
}
return closestPlayer;
}
//
// Trova la distanza sul terreno (ignorando l'altezza)
// tra due Location
//
public double distance(Location loc1, Location loc2) {
  return Math.sqrt(
    Math.pow(loc1.getX() - loc2.getX(), 2) +
    Math.pow(loc1.getZ() - loc2.getZ(), 2)
  );
}
```

```
// Ti fa esplodere
public void explode() {
  plugin.cowDied(cow); // Avvisa la classe genitore
  Location cowLoc = cow.getLocation();
  cow.getWorld().makeExplosion(cow,
        cowLoc.getX(), cowLoc.getY(), cowLoc.getZ(),
        3.0f, true);
  removeMe();
}

// Fatto, perché il chunk è stato eliminato o per l'esplosione
public void removeMe() {
  cow.kill();
  Canary.getServer().removeSynchronousTask(this);
}

// Fa saltare questa mucca verso il bersaglio
public void jump(Location target) {
  Location cowLoc = cow.getLocation();
  double multFactor = 0.075;
  Vector3D v = new Vector3D(
    (target.getX() - cowLoc.getX()) * multFactor,
    0.8,
    (target.getZ() - cowLoc.getZ()) * multFactor
  );
  cow.moveEntity(v.getX() + (Math.random() * -0.1),
    v.getY(),
    v.getZ() + (Math.random() * -0.1));
}

// Callback per avviare ed eseguire il corpo dell'attività
public void run() {
  if (cow.isOnGround()) { // altrimenti continua a saltare
    Location cowLoc = cow.getLocation();
    Player p = cow.getWorld().getClosestPlayer(cow,  10000);
    if (p == null) {
      return;
    }
    Location pLoc = p.getLocation();
    double dist = distance(cowLoc, pLoc);

    if (dist <= 4) {
      explode();
    } else if (dist <= 200) {
      jump(pLoc);
    }
   }
  }

}
```

Potresti aver notato che non è implementato nulla di quanto ho citato nel mio elenco. Per esempio, non c'è alcun controllo sul numero eccessivo di mucche, e non uccido direttamente il giocatore bersaglio. Il fatto che abbia pensato di aver bisogno di questi elementi non significa che debba averli già scritti. Posso provvedere successivamente, se necessario.

Potrei anche incappare in altri problemi che non avevo considerato. Per esempio, cosa accade alla mucche quando il server si ferma? Le mucche rimarranno nel mondo, ma non saranno più `CreeperCow` perché il plug-in non ne tiene

traccia. Forse dovrei far sparire le mucche alla chiusura. O magari avere delle mucche in più non è un vero problema? Ah, il software.

Prova da solo

Questo è stato il nostro viaggio con il plug-in `CreeperCow`. Il tuo viaggio con il tuo plug-in sarà probabilmente diverso. Prova comunque a seguire questi passi: identifica le parti che ti servono, magari usando le schede CRC, e poi scrivi in italiano la sequenza di quello che dovrà accadere. Prendi il tutto e crea le funzioni, passa i dati che ti servono e memorizza quelli che devi ricordare, senza dimenticare di aggiungere comandi per i test in modo da verificare ogni parte singolarmente.

Non dimenticare di rimuovere i comandi di test prima di dare il plug-in ai tuoi amici. Oppure, se vuoi esagerare, impostalo in modo tale che serva uno speciale permesso da "Sviluppatore" per eseguire i test, e assegnatelo.

Fatto! Adesso la tua toolbox è completa.

La tua toolbox è completa!

100%

Adesso sai come:

- Usare la shell a riga di comando.
- Utilizzare il compilatore Java (`javac`).
- Eseguire un server di Minecraft.
- Distribuire un plug-in.
- Connetterti a un server locale.
- Usare le variabili di Java per numeri e stringhe.
- Usare le funzioni di Java.
- Usare le istruzioni `if`, `for` e `while`.
- Usare gli oggetti Java.
- Usare le istruzioni `import` per i package Java.
- Usare `new` per creare gli oggetti.
- Aggiungere un nuovo comando a un plug-in.
- Lavorare con gli oggetti `Location`.
- Trovare blocchi ed entità.
- Usare le variabili locali.
- Usare le variabili globali a livello di classe.
- Usare gli `ArrayList`.
- Usare le `HashMap`.
- Usare `private` e `public` per controllare la visibilità.
- Modificare i blocchi di Minecraft.
- Modificare e generare le entità.
- Monitorare gli eventi di gioco e rispondere.
- Gestire i permessi dei plug-in.
- Creare una classe separata.
- Programmare un’attività che deve essere svolta nel futuro.

- Programmare un’attività in modo che venga svolta in continuazione.
- Salvare e caricare i dati di configurazione.
- Costruire un codice complesso partendo da funzioni più semplici.
- Salvare e caricare i dati di gioco.
- Usare gli oggetti `DataAccess` per lavorare con il database.
- Catturare e sollevare eccezioni in Java.
- Usare Git per tenere traccia delle modifiche al codice.
- Tornare a versioni precedenti del codice (come se si premesse un pulsante *Annulla*).
- Mantenere più versioni del codice contemporaneamente.
- Eseguire un backup del codice sul cloud.
- Usare schede CRC per stabilire classi e responsabilità.
- Tradurre le responsabilità in funzioni.
- Eseguire progressivamente dei test.

È solo l’inizio

È stato un giro divertente, ma abbiamo appena scalfito la superficie. C’è molto di più da sapere su Java e su Canary di quanto abbiamo trattato qui. Inoltre c’è tanto da imparare sulla programmazione in generale, cose che scoprirai man mano.

Spero tuttavia che per te sia stato divertente. Non fermarti adesso! Procurati un altro paio di libri su Java o su un altro linguaggio, sulla programmazione, sul web design e via di seguito. Non smettere mai di leggere e studiare.

E quando avrai creato qualcosa di davvero eccezionale, mandami un’e-mail per farmelo sapere.

Grazie per aver acquistato questo libro, e auguri!

Andy

andy@pragprog.com

Come leggere i messaggi di errore

I messaggi di errore prodotti dal compilatore di Java, dal sistema e dal server di Minecraft solitamente sono abbastanza chiari, ma non sempre. Il compilatore javac, in particolare, ogni tanto va in confusione, e genera messaggi che non sono di grande aiuto.

Leggi questa appendice anche se non hai ottenuto un errore, perché alcune delle informazioni contenente in queste pagine potrebbero aiutarti a decifrare altri messaggi che qui non sono trattati.

Ho incluso alcuni dei messaggi di errori più comuni che potresti incontrare, con alcune osservazioni e commenti. Se incappi in un errore incomprensibile che non è tra quelli spiegati qui, prova a cercare con Google il testo dell'errore: è molto probabile che qualcun altro abbia avuto lo stesso problema, e puoi trarre vantaggio dalla sua esperienza.

Messaggi di errore del compilatore di Java

I messaggi di errore del compilatore di Java hanno solitamente questo aspetto:

```
src/helloworld/HelloWorld.java:21:  cannot find symbol
symbol  : class MessageReceiver
location: class helloworld.HelloWorld
      public void helloCommand(MessageReceiver caller, String[] parameters)
```

Java sta cercando di dirti esattamente dove l'errore si è verificato e quale pensa sia il problema.

Il primo segmento di testo è il nome del file dove Java ritiene si trovi l'errore, in questo caso il file `src/helloworld/HelloWorld.java`. Poi c'è un numero tra due `:`, che indica il numero della riga (qui è `21`). A seguire c'è il messaggio vero e proprio, `cannot find symbol`. Dopo di questo trovi i dettagli specifici, che in questo esempio è il simbolo che javac non riesce a trovare, e altre informazioni sulla posizione del simbolo mancante.

Quindi, alla riga `21` di `HelloWorld.java`, Java non sa nulla dell'oggetto `MessageReceiver`. Vediamo le possibili cause.

javac: "cannot find symbol"

"Cannot find symbol" significa che il compilatore ha incontrato una parola o una parte del testo, che non capisce.

Questo errore può dipendere da diversi problemi. Per esempio, cosa succede se aggiungi un'istruzione di assegnazione come `i = 10` senza dichiarare cos'è `i`?

```
src/helloworld/HelloWorld.java:23:  cannot find symbol
symbol  : variable i
location: class helloworld.HelloWorld
        i = 10;
        ^
```

Il compilatore non ha idea di cosa sia `i` o di cosa dovrebbe essere, quindi si lamenta.

Per correggere l'errore puoi aggiungere una dichiarazione come `int i;` sopra a questo codice o sulla stessa riga di `int i = 10;`. Così il compilatore capisce che `i` è una variabile locale.

E se avessi dichiarato la variabile ma ottenessi comunque un errore? Per

esempio, nella chiamata a `helloCommand`, sto dichiarando un parametro `MessageReceiver caller`. `caller` è la mia variabile, di tipo `MessageReceiver`, ma ricevo lo stesso errore:

```
src/helloworld/HelloWorld.java:21:  cannot find symbol
symbol  : class MessageReceiver
location: class helloworld.HelloWorld
      public void helloCommand(MessageReceiver caller, String[] parameters)
```

L'errore può indicare un'istruzione di importazione mancante o scritta in modo sbagliato. Il compilatore sa che `caller` è una variabile di tipo `MessageReceiver`, ma non sa cos'è un `MessageReceiver`.

In questo caso, basta aggiungere

```
import net.canarymod.chat.MessageReceiver;
```

in cima al file per correggere l'errore. Leggi l'Appendice F per un elenco delle istruzioni di importazione più comuni che abbiamo usato nel libro, oppure consulta la documentazione di Java o di Canary.

javac: manca il punto e virgola

```
src/helloworld/HelloWorld.java:6:  ';' expected
import net.canarymod.chat.MessageReceiver
```

Abbiamo dimenticato il punto e virgola alla fine della riga di importazione, che dovrebbe essere così:

```
import net.canarymod.chat.MessageReceiver;
```

Tuttavia, questo messaggio di errore può essere ambiguo. Per esempio, supponi di aver dimenticato la parentesi graffa di apertura (`{`) in un blocco di codice:

```
public void disable()
  log.info("Stopping.");
}
```

Otterrai una serie di errori, partendo da quello relativo al punto e virgola mancante e così via per diverse righe di codice, al di là dell'errore iniziale:

```
src/helloworld/HelloWorld.java:18:  ';' expected
  public void disable()
                       ^
src/helloworld/HelloWorld.java:21:  class, interface, or enum expected
  public void helloCommand(MessageReceiver caller, String[] parameters)
         ^
...
```

Il compilatore è confuso: pensa che avrebbe dovuto esserci un punto e virgola dopo `onDisable()`, ma in realtà quello che serviva era una `{`. Non riesce a capire cosa avrebbe dovuto succedere e inizia a lamentarsi del fatto che *l'intero file* non è

come dovrebbe.

Ecco perché in molti casi è sufficiente leggere le prime due o tre righe degli errori e ignorare il resto, perché molti degli errori segnalati successivamente spariscono non appena si corregge il primo.

javac: “illegal start of expression”

“Inizio dell’espressione non valido” è un messaggio di errore generico che viene prodotto quando abbiamo scritto un’espressione un po’ disordinata e il compilatore non riesce ad avviarla. Per esempio, supponiamo di dimenticare la parentesi graffa di chiusura (`}`) alla fine di una funzione:

```
public void disable() {
  log.info("Stopping.");
```

Anche in questo caso otteniamo una serie di errori, partendo dalla parentesi mancante e proseguendo per il resto del file:

```
src/helloworld/HelloWorld.java:21:  illegal start of expression
  public void helloCommand(MessageReceiver caller, String[] parameters)
  ^
src/helloworld/HelloWorld.java:21:  ';' expected
  public void helloCommand(MessageReceiver caller, String[] parameters)
                          ^
src/helloworld/HelloWorld.java:21:  ';' expected
  public void helloCommand(MessageReceiver caller, String[] parameters)
                                                            ^
src/helloworld/HelloWorld.java:21:  not a statement
```

Ricorda che “inizio non valido” in realtà significa “non scritto nel modo corretto”.

javac: la classe è pubblica, dovrebbe essere dichiarata in un file chiamato…

Nel mio `HelloWorld.java`, sono stato creativo e aggiunto un’altra dichiarazione di classe alla fine del file:

```
public class TooMany {
  // ...
}
```

La mia iniziativa ha generato un messaggio di errore molto descrittivo:

```
src/helloworld/HelloWorld.java:32:  class TooMany is public,
should be declared in a file named TooMany.java
public class TooMany {
```

Ricorda che ogni classe pubblica deve trovarsi in un file separato che ha lo

stesso nome della classe, contenuto a sua volta in una cartella che ha lo stesso nome del package.

Tuttavia, puoi dichiarare una classe `private` all'interno della tua classe pubblica nello stesso file. Può rivelarsi utile per le classi helper più piccole.

javac: "incompatible types"

Nella versione del plug-in `BackCmd` in cui abbiamo salvato i giocatori su disco, ho commesso un errore: `playerTeleports` restituisce uno `Stack` di oggetti `Location`, ma inavvertitamente ho cercato di assegnarlo a uno stack di oggetti `Player`, così:

```
Stack<Player> locs = playerTeleports.get(player.getName());
```

Il risultato è un messaggio di errore chiarissimo, "i tipi sono incompatibili":

```
src/backcmdsave/BackCmdSave.java:98:  incompatible types
found   : java.util.Stack<net.canarymod.api.world.position.Location>
required: java.util.Stack<net.canarymod.api.entity.living.humanoid.Player>
        Stack<Player> locs = playerTeleports.get(player.getName());
```

Java dice che ha trovato una `Location` dove si aspettava di dover usare un `Player`.

In casi come questi, può essere di aiuto pensare come il computer, qui come il compilatore. Immagina di essere il compilatore di Java. Hai appena finito di leggere una riga di codice fino al punto e virgola e stai iniziando la riga successiva. Vedi la prima parte:

```
Stack<Player> locs =
```

Ah! Il programmatore sta dichiarando una variabile chiamata `locs`, che è uno `Stack` generico di oggetti `Player`, a cui assegneremo un valore iniziale.

Il compilatore vede poi il resto dell'istruzione:

```
playerTeleports.get(player.getName());
```

Il compilatore (cioè tu) prende `playerTeleports.get()` e vede che restituisce uno `Stack` di oggetti `Location`, cosa che non va fatta. Il programmatore sta dicendo che abbiamo uno `Stack` di oggetti `Player`, ma il codice sta cercando di assegnare uno `Stack` di oggetti `Location`. Javac si aspettava un `Player` e invece ottiene una `Location`.

Il compilatore ovviamente può sbagliare. Nessun compilatore può dedurre le tue intenzioni. Al massimo può valutare quello che hai effettivamente fatto, non quello che intendevi fare.

Ricordalo sempre quando cerchi di decifrare i suoi messaggi di errore: sono il

risultato del suo punto di vista, e lui non è in grado di leggerti nel pensiero (almeno non ancora, ma ci sta lavorando).

Per chiudere, basta correggere la corrispondenza di tipo dello `Stack` per risolvere il problema:

```
Stack<Location> locs = playerTeleports.get(player.getName());
```

Messaggi di errore del server di Canary

Il server di Canary visualizza i messaggi di errore nel suo log (il registro, `~/Desktop/server/logs/latest.log`) o direttamente nella console di gioco di Minecraft mentre stai giocando. Vediamo gli errori più comuni che potresti incontrare con un nuovo plug-in.

Log del server: "could not load"

Molti degli errori nel file di registro o nella console del server (durante la sessione di gioco) sono espressi in modo più chiaro che non i messaggi del compilatore. Uno degli errori più critici e comuni è questo:

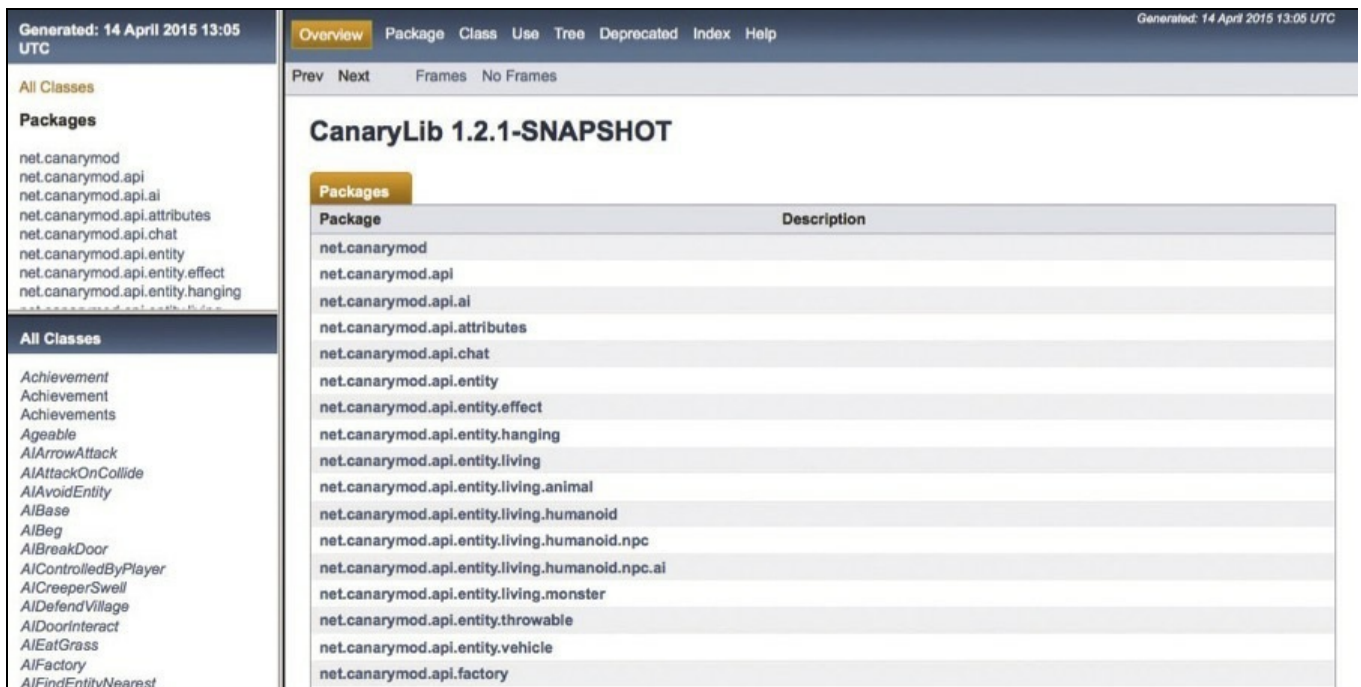
```
[SEVERE] Could not load 'plugins/HelloWorld.jar' in folder 'plugins'
net.canarymod.exceptions.InvalidPluginException:
java.lang.ClassNotFoundException:  helloworld.HelloWorld
```

Hai scritto un nuovo plug-in, e ti compare un errore che dice che non può essere caricato. In genere è segno che il nome del package e quello del plug-in specificati nel file `Canary.inf` non coincidono con il nome del package e della classe nel codice. Controlla se hai digitato male qualcosa. Ricorda che il nome di un package per convenzione è scritto tutto in minuscolo, mentre quello di una classe è un misto di maiuscole e minuscole.

Console di Minecraft: "unknown command"

Se il plug-in si compila bene ma Minecraft ti segnala un errore "comando sconosciuto", controlla l'annotazione `@Command()` per essere certo di aver scritto il comando nel modo giusto e di aver specificato correttamente un numero `min` (minimo) di argomenti o un'impostazione dei permessi (`permission`).

Come leggere la documentazione di Canary

Ti capiterà spesso di dover leggere direttamente la documentazione di Canary per capire come fare qualcosa nel mondo di Minecraft riguardo a un Block, un Player, un Ocelot, una Cow, un Creeper e altro. Devi capire quali classi utilizzare e quali funzioni queste offrono. La documentazione elenca tutte queste informazioni; vediamo come trovarle.

La documentazione JavaDoc di Canary

La documentazione JavaDoc di Canary è focalizzata sulle classi di sistema, elencate con i relativi package. Trovi quanto segue.

- Il nome del package, che è quello che userai nell'istruzione di importazione (leggi il Capitolo 2 e l'Appendice F sull'argomento).
- Il nome della classe, che è quello che usi per dichiarare le variabili e creare gli oggetti con `new` (leggi i Capitoli 4 e 5).
- Le funzioni (i metodi) nella classe che puoi eseguire con `()` e dei parametri (leggi il Capitolo 7).
- Le classi genitore o le interfacce utilizzate, comprese eventuali funzioni aggiuntive che puoi chiamare (leggi il Capitolo 5).

Accedi con il tuo browser alla documentazione di Canary all'indirizzo `https://ci.visualillusionsent.net/job/CanaryLib/javadoc`, e tutti quei tesori nascosti saranno tuoi!

Figura B.1 JavaDoc di Canary.

Ma cosa significa tutta questa roba? E dove trovi quello che ti serve?

A sinistra nello schermo c'è un elenco di tutti i nomi dei package di alto livello

presenti in Canary, partendo da `net.canarymod`.

Ci sono poi alcuni link. Fai clic su un link, e vedrai apparire tutti i “figli” di quell’elemento. Fai clic su un package nell’angolo in alto a sinistra, per esempio su `net.canarymod.api.world`, e tutte le classi e quanto contenuto in quel package compariranno nel riquadro in basso a sinistra.

Fai clic su *World* nel riquadro in basso a sinistra, e vedrai tutte le funzioni (i metodi) che puoi usare in un mondo. Fai clic su qualsiasi metodo o nome di classe per avere i dettagli.

La documentazione JavaDoc di Oracle

Se stai cercando informazioni più generali sulle classi di Java (e non solo su quelle specifiche di Canary), vai con il tuo browser alla documentazione di Oracle all'indirizzo `http://docs.oracle.com/javase/7/docs/api`.

La pagina che ti appare è molto simile a quella del sito di Canary (Figura B.2).

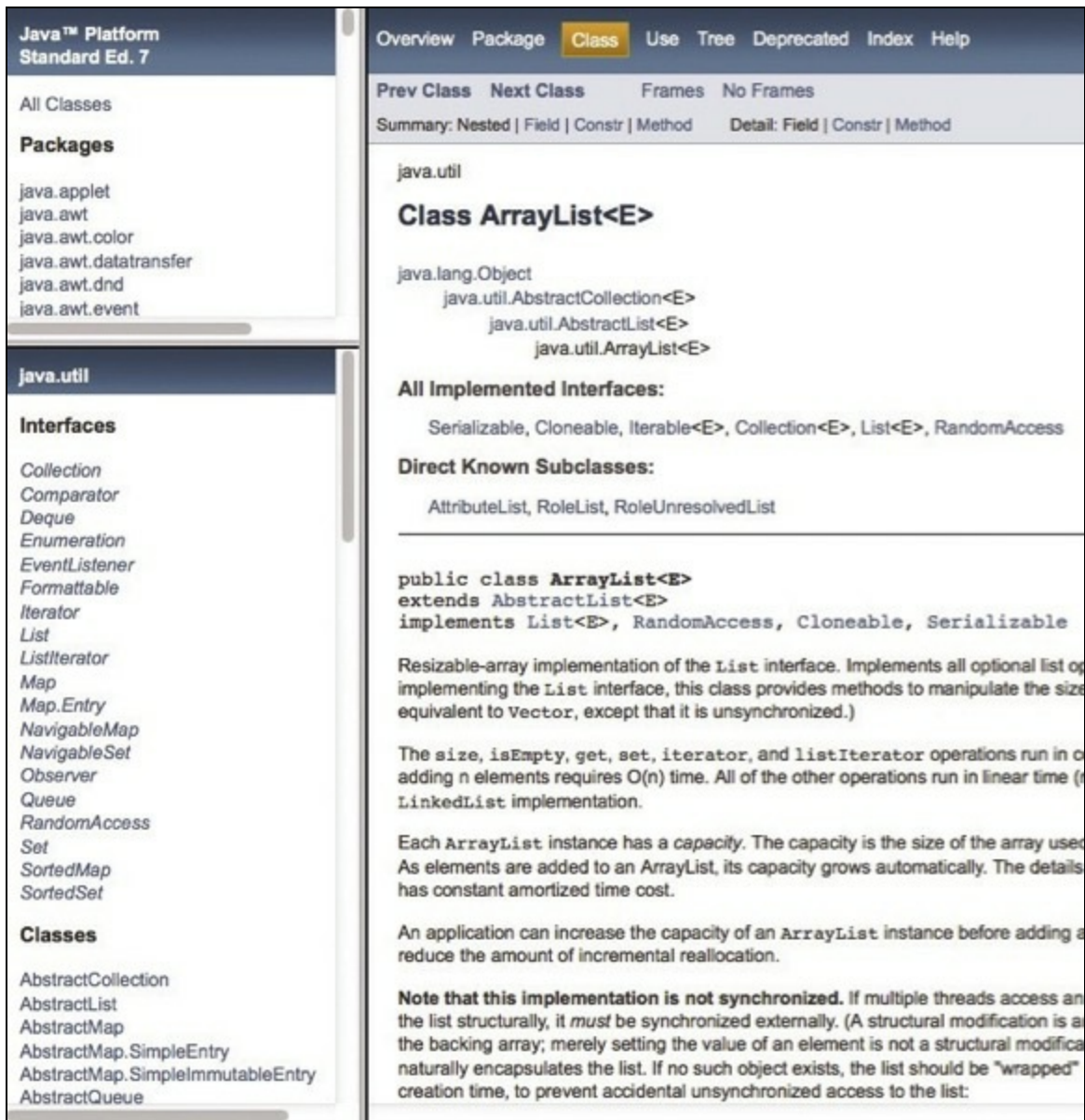



Figura B.2 JavaDoc per Java (Oracle).

Nell'angolo in alto a sinistra trovi un elenco di package che puoi scorrere, compresi i più utili come `java.util` e `java.math`. Sotto trovi un elenco di interfacce (come le classi ma senza le rispettive funzioni) e le classi contenute in quel

package. Il riquadro di destra contiene la documentazione relativa alla classe su cui hai fatto clic. Qui ho scelto il package `java.util` e la classe `ArrayList`.

La documentazione sulle classi è eccezionale quando sai a grandi linee quello che vorresti ma ti servono i dettagli. Ma se proprio non sapessi da che parte cominciare?

Wiki e tutorial

Fortunatamente questo libro ti ha spiegato dove cercare le specifiche per i `Blocks` di `BlockType` diversi, e quali sono i vari tipi di oggetti `Entity`. Ma se ti interessasse qualcosa che non abbiamo trattato?

Oltre alla documentazione, il sito di Canary contiene alcuni tutorial e spiegazioni dettagliate su vari argomenti (http://canarymod.net/books). Trovi anche un forum dove puoi porre domande e chiedere aiuto (http://canarymod.net/forum).

E non dimenticarti del forum sul sito web di Pragmatic Bookshelf, dove puoi fare domande e inviare commenti sul libro, oltre a diffondere tutte le scoperte esaltanti che hai compiuto leggendo queste pagine (https://forums.pragprog.com/forums/382).

Come installare un server desktop

Adesso che hai dei bellissimi plug-in nuovi e personalizzati che girano sul server sul tuo computer, vuoi che i tuoi amici possano connettersi e giocare.

Per farlo hai due modi: impostare il tuo computer personale in modo che altri possano accedervi o impostare un server permanente nel cloud. (Leggi l'Appendice D sull'argomento.)

Nel primo caso, ricorda che il computer dovrà essere acceso e il server di Minecraft connesso. Se ti va bene, procedi come segue.

Per iniziare devi affrontare due questioni:

- ti serve un meccanismo che permetta ai tuoi amici di trovare l'indirizzo della tua macchina in Internet;
- devi tenere aperta al mondo la porta di Minecraft, che è la 25565.

Il modo più semplice per raggiungere lo scopo è utilizzare un software specifico che si occupi di tutto questo per te. In alternativa puoi lavorare direttamente su ogni punto della procedura, che è molto più divertente ma anche molto più impegnativo.

La strada più facile: LogMeIn

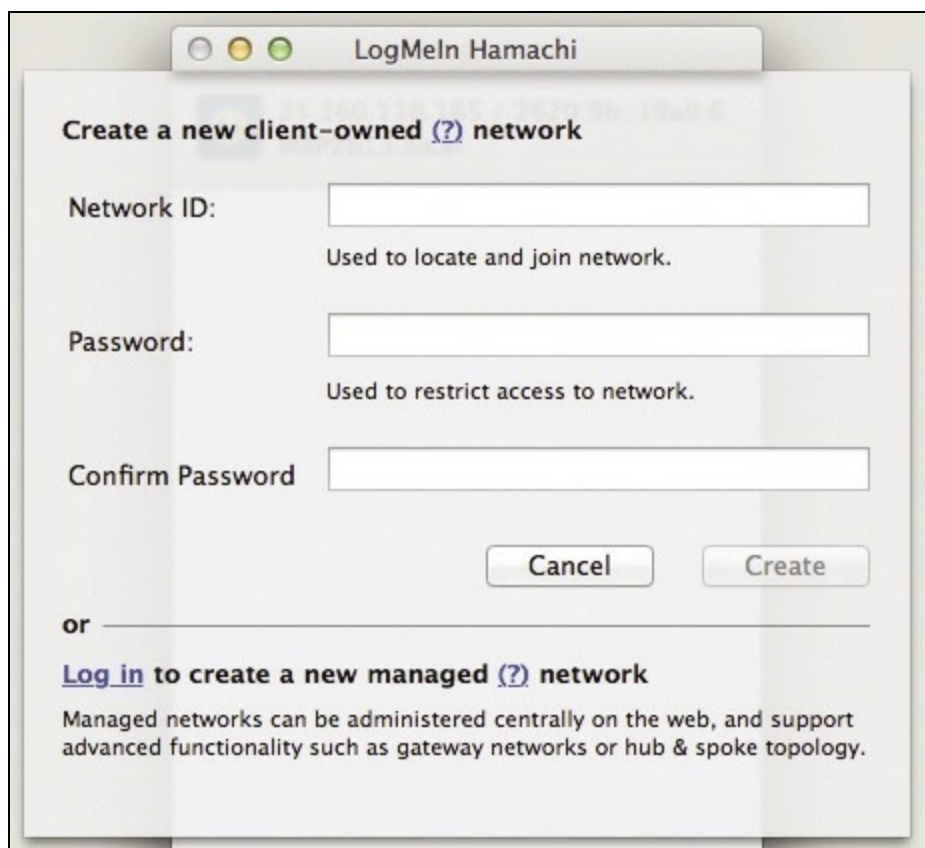
Hamachi è un prodotto di LogMeIn disponibile in una versione gratuita (*Non gestita*) per un massimo di cinque utenti (https://secure.logmein.com/products/hamachi/download.aspx) e in una versione completa e a pagamento (*Gestita*) per computer Windows e OS X.

Hamachi genera un indirizzo IP a cui i tuoi amici possono connettersi. Non devi trafficare localmente con il firewall, il router o a altro; funziona subito. Per impostarlo, scarica la versione *Non gestita* e installala.

COS'È UNA PORTA?

In questa appendice e nella prossima troverai dei riferimenti alle porte di rete e alla porta predefinita di Minecraft, la 25565. Una porta è un numero convenzionale che consente ai computer di comunicare in una rete. Per esempio, i browser usano le porte 80 e 443 per parlare con i server web.

Quando avvii LogMeIn per la prima volta, ti chiederà di unirti a una rete o di crearne una:

Creiamone una. Io ho chiamato la mia `andys_minecraft_server`:

Siamo operativi!

Ora chiedi ai tuoi amici di scaricare lo stesso software e di unirsi alla tua rete.

Facendo clic con il tasto destro del mouse sul nome della tua rete, dovrebbero poter selezionare *Copy IPv4 address* e usare questo indirizzo nel client di Minecraft per connettersi a te.

Se vuoi che i tuoi amici non siano più connessi, fai clic sull'icona dell'interruttore per disattivare la rete. Fatto.

La strada più difficile: a mano

Impostare il computer in modo che sia aperto e rintracciabile in Internet senza l'aiuto di strumenti come LogMeIn richiede impegno. Ecco alcuni dei vantaggi della soluzione più difficile.

- Non ti costa niente.
- Puoi avere la connessione con più di cinque utenti.
- Sei disponibile per tutti, non solo per i tuoi amici.
- È un lavoro utile e interessante.

Prima, però, devi sapere qualcosa su come funzionano le reti e Internet. Qui riporto alcune dei passi più importanti, ma potresti dover leggere dell'altro. Soprattutto, non sarò in grado di darti delle istruzioni precise, perché ogni router o modem è diverso. Puoi trovare informazioni online passando da Google o su siti specifici come `http://portforward.com` o `http://www.navigaweb.net/2007/08/configurare-il-port-forwarding-su-un.html`.

Il primo passo da fare è permettere ai tuoi amici di trovare il tuo computer.

IP statico e DNS dinamico

Un indirizzo IP è un numero che chiunque nel mondo può usare per connettersi a un server. Solitamente è costituito da quattro serie di numeri separate da un punto, come 93.184.216.119 (`example.com`) o 127.0.0.1 (la tua macchina locale, `localhost`).

È il molto probabile che il tuo provider di servizi internet (ISP) modifichi spesso il tuo indirizzo IP, quindi non puoi essere certo di avere lo stesso indirizzo tutti i giorni, a meno che non paghi per avere un *IP statico*. Se già sapevi di poter chiedere un IP statico, forse sai anche come impostare un nome e un server DNS per puntare alla tua macchina (se invece non lo sai e ti interessa un IP statico, leggi l'Appendice D).

Tuttavia, non ti serve un IP statico perché i tuoi amici ti trovino. C'è un trucco che ti permette di usare altrettanto facilmente il tuo indirizzo IP dinamico che cambia sempre.

Per iniziare, se non conosci che aspetto ha il tuo IP, vai all'URL `http://www.checkip.org`. Lì vedrai come appare al mondo esterno.

Se dovesse cambiare ogni due o tre giorni o ore, però, sarebbe fastidioso, poiché ti costringerebbe a comunicare continuamente ai tuoi amici il nuovo indirizzo IP da usare per connettersi a te. Ma sei fortunato! Puoi risolvere tutto registrando gratuitamente (o a poco prezzo) un DNS dinamico su siti come `http://dyndns.org` o `http://www.noip.com`.

In questo modo i tuoi amici potranno usare un nome più "descrittivo" per trovarti in Internet, qualcosa come `andyminecraft.dns.org`.

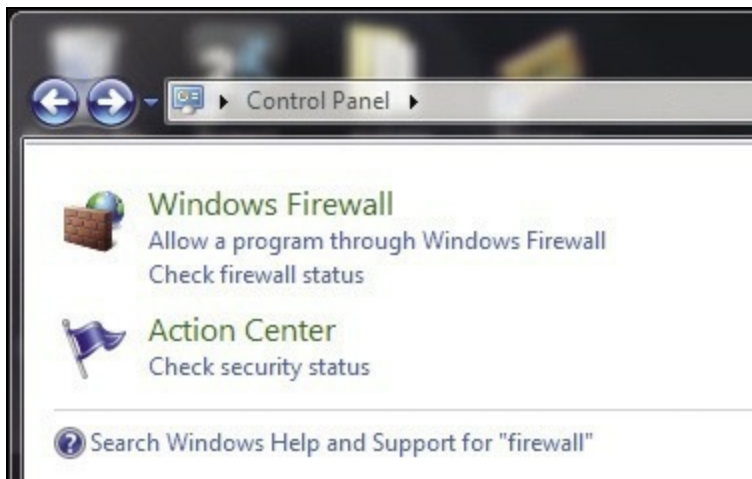
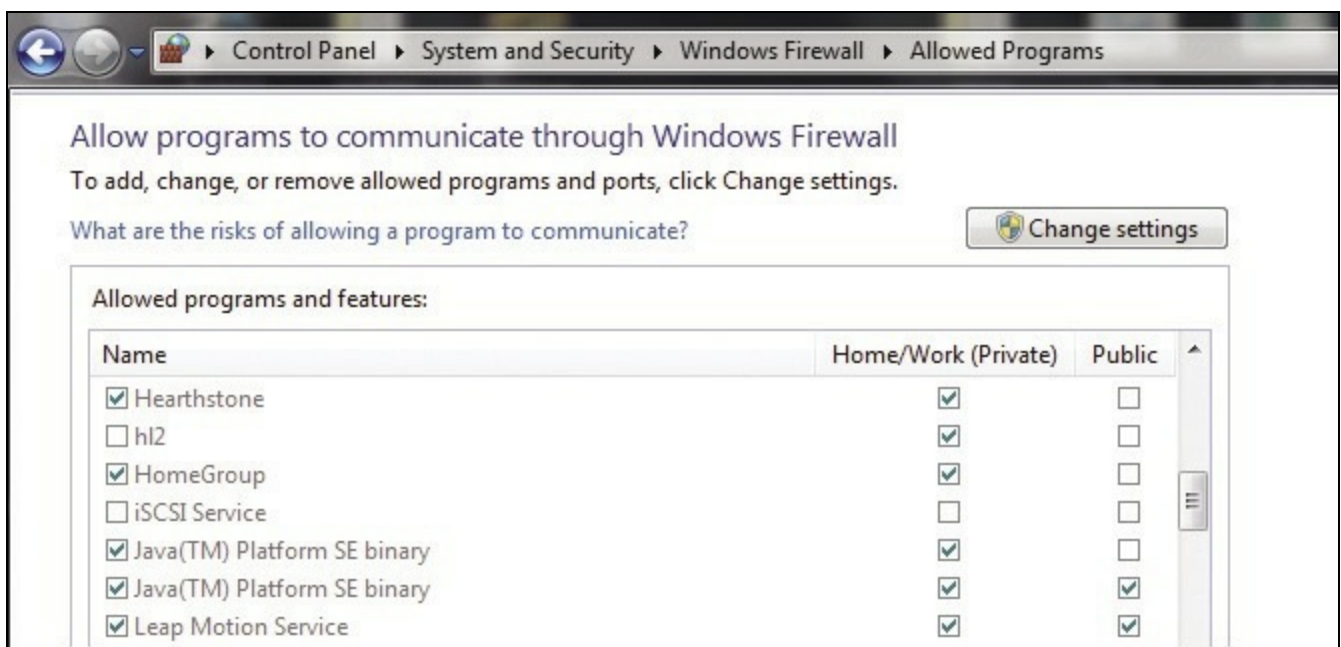
Ora che sanno come trovare la tua macchina, devi farli entrare.

Aprire il firewall

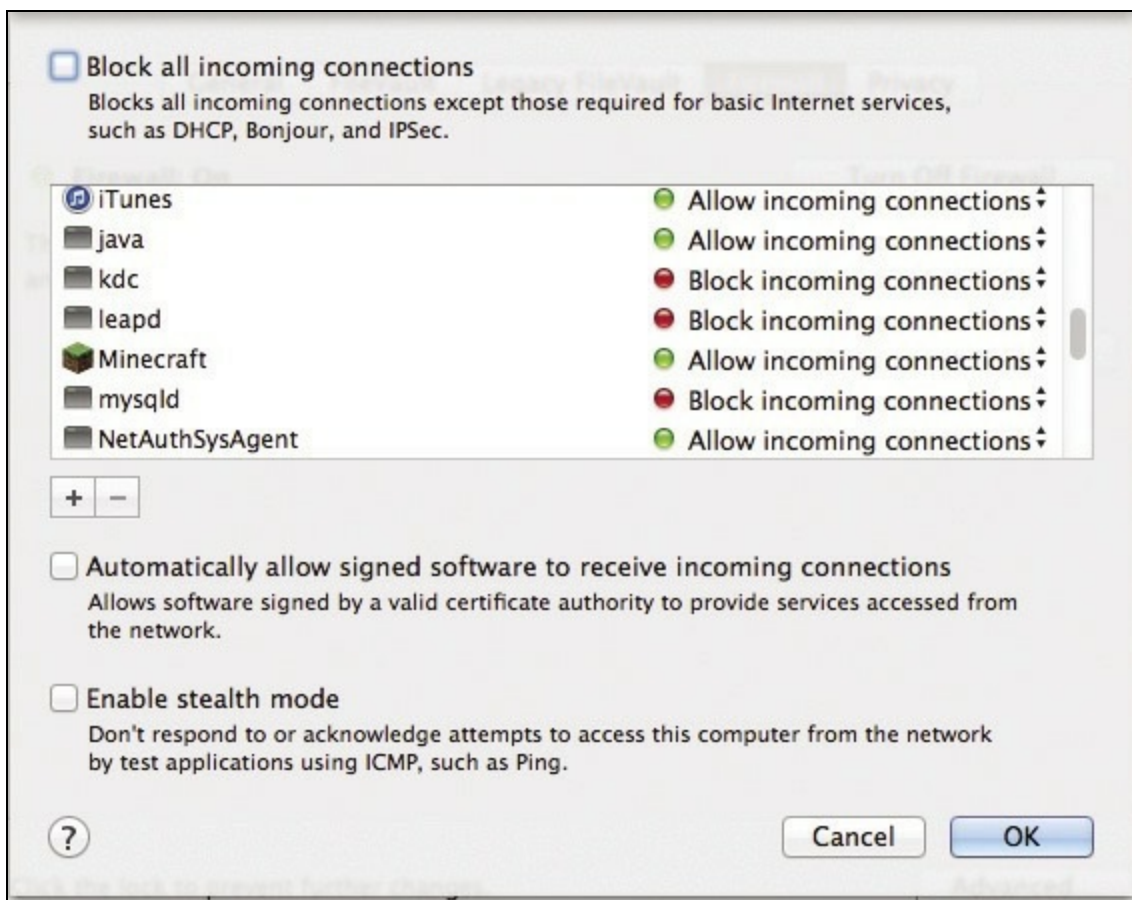
I computer Windows, OS X e Linux, così come il modem o il router dell'ISP, eseguono tutti un software firewall, che ha lo scopo di bloccare proprio quello che stiamo cercando di fare qui. Il concetto è che ci sono in giro dei malintenzionati che tentano continuamente di attaccare la tua macchina e di entrarci. I computer alzano allora un *firewall* (una barriera) per bloccare tutte le porte a eccezione di quelle che vuoi effettivamente lasciare aperte e in attesa di connessione.

Devi quindi accertarti che il tuo computer apra la porta 25565, che è quella di Minecraft. Come farlo dipende dal sistema operativo su cui lavori.

Aprire il firewall in Windows

In Windows 8, digita `Firewall`, e Metro ti guiderà attraverso la procedura *Consenti ai programmi di comunicare con Windows Firewall*. Consulta anche la documentazione online di Windows per ulteriori informazioni (`http://windows.microsoft.com/it-it/windows7/products/features/windows-firewall`). In Windows 7, apri il *Pannello di controllo* e seleziona *Windows Firewall* > *Consenti ai programmi di comunicare con Windows Firewall*. (Qui vedi la schermata del mio computer, che è in inglese.)

Poi abilita l’accesso pubblico per Java:

Aprire il firewall su OS X

Apri *System Preferences* (*Preferenze di sistema*) e poi il pannello *Security & Privacy* (*Sicurezza e Privacy*).

Se il firewall non è attivo, sei a posto. Se invece è attivo, assicurati che Java abbia i permessi per consentire le connessioni in entrata:

Aprire il firewall in Linux

Se hai impostato un firewall usando IPTables, controlla che la porta 25565 sia

aperta.

Per tutti i sistemi operativi

Aprire il firewall sul computer potrebbe non bastare. Per verificare se Internet può accedere alla tua porta, vai sul sito http://canyouseeme.org. Digita la porta da controllare, per esempio la 25565, e fai clic su *Check Port*. Se funziona, puoi partire, altrimenti devi fare un'altra cosa.

Se sei connesso a Internet tramite un router o un modem, allora anche *quel* dispositivo potrebbe avere un firewall. Non solo; anche se questo ha la porta aperta a Internet, dovrai "inoltrare" (*forward*) dal router al tuo computer il traffico che passa attraverso quella porta.

Port forwarding

Quando modifichi le impostazioni del router devi procedere con cautela. Se cambi inavvertitamente qualcosa che non avresti dovuto toccare, potresti finire con il danneggiare il dispositivo e perdere l'accesso a Internet. Attieniti alle indicazioni che trovi qui, quindi. Tuttavia, ogni router è diverso, quindi non posso fornirti istruzioni troppo specifiche. Questi sono comunque i fondamenti.

1. Collegati al router. Potresti già conoscere il suo nome utente e la password. Se così non fosse, il router potrebbe avere un nome utente e una password predefiniti che puoi cercare in Google.
2. In qualche menu dovrebbe esserci una sezione relativa al port forwarding. Nel mio router si trova sotto *Gateway* e sotto a *Forwarding*.
3. Devi dire al router di inoltrare la porta 25565 (sia la TCP sia la UDP, se te lo chiede) dal mondo esterno (il tuo IP esterno, come riportato in http://www.checkip.org) all'IP interno del tuo computer.

Il computer ha un indirizzo IP interno che viene usato per le comunicazioni con il router/modem. Per vedere qual è questo indirizzo, puoi eseguire un comando sul tuo computer.

Windows

Esegui `ipconfig`, e cerca *IPv4 address* sotto a *Ethernet adapter* o a *Wireless LAN*.

OS X/Linux

Esegui `ifconfig`. Probabilmente si trova sotto a *inet* e poi a *en1*.

Puoi ricorrere a Google per trovare ulteriori informazioni sull'impostazione del tuo router, oppure vai su un sito come http://portforward.com.

Un'ultima complicazione: il tuo indirizzo IP interno potrebbe cambiare molto spesso, proprio come quello dell'ISP. Se è un server DHCP a fornirti l'indirizzo IP sul router, puoi chiedergli di creare anche una *prenotazione DHCP* in modo da poter ottenere sempre lo stesso indirizzo IP interno. Oppure puoi modificare direttamente il port forwarding quando o se questo indirizzo cambia.

Se i tuoi amici si collegano solo ogni tanto e vuoi evitare tutte le seccature legate all'impostazione, questa è la strada ideale da seguire. Se invece vuoi che si colleghino più persone e che il tuo server sia attivo 24 ore su 24, sette giorni su sette, dovrai impostare un server sul cloud, come descritto dettagliatamente nell'Appendice D.

QUALITY OF SERVICE

Molti router moderni hanno una funzione dove puoi specificare la *Quality of Service* (QoS, cioè la qualità del servizio) che desideri per le varie porte/servizi. L'idea è che tu possa impostare una priorità alta o bassa per ogni tipo di traffico di rete.

Può essere utile quando esegui un server di Minecraft server locale, dove puoi impostare una proprietà bassa per la porta del gioco. Abbassando la priorità di Minecraft al minimo avrai la certezza che il tuo traffico avrà la precedenza su quello dei tuoi amici che usano il server.

Come installare un server sul cloud

Se vuoi, puoi eseguire un server di Minecraft sul tuo computer, ma questa soluzione ha alcuni svantaggi. Quello più importante è che il tuo computer deve sempre rimanere acceso e connesso a Internet, 24 ore al giorno, sette giorni su sette. Su un portatile questo è un po' difficile.

Quello che stai facendo sul computer può rallentare il server di Minecraft e i tuoi giocatori, e quanto farà il server rallenterà gli altri programmi sul tuo computer. Se sai che molte persone vogliono connettersi al tuo server e giocare, puoi pensare di impostare un server remoto sul cloud.

Impostare un server di Minecraft sul cloud è simile a impostare un server localmente, ma con una differenza non da poco: non hai l'accesso fisico al computer che sta eseguendo il tuo server di Minecraft. Questo significa niente tastiera, niente schermo, nessun interruttore o tasto di riavvio. Ma non preoccuparti: sai come usare la riga di comando, ed è tutto quello che ti serve.

Vediamo cosa significa e come sfruttarlo.

Cos’è il cloud?

Qualcuno sostiene che “il *cloud*” (la nuvola) non sia altro che un grosso armadio pieno di computer in Virginia. La cosa non è troppo lontana dalla verità, visto che Amazon e altri mantengono grossi centri dati in quella zona, oltre che in California e in molti altri luoghi nel mondo. “Il *cloud*” è in effetti un gruppo di computer da qualche parte in Internet.

Quando le persone parlano di computer sul cloud, intendono un computer a cui si accede solo da Internet, una macchina di cui non si conosce la posizione fisica e che qualcuno possiede e mantiene. Come vedi, è una definizione piuttosto generica e flessibile. I servizi cloud disponibili variano, così come variano le tariffe, l’affidabilità e quello che ottieni con la cifra che paghi. I tipi di servizio che ti potrebbero interessare di più sono due.

- Servizi specifici per Minecraft: il provider effettua tutto il lavoro di manutenzione e amministrazione del server. Caricherà o fornirà i plug-in, bloccherà gli eventuali tentativi di attacco e farà girare il server. Tu non devi fare niente. Invece di pagare una tariffa in base alla dimensione del server, potresti dover pagare in base al numero di giocatori che sono online contemporaneamente.
- Servizi generici: il provider fornisce un sistema operativo e il login, ma nient’altro: sei tu a dover installare Java e Canary, caricare i plug-in, amministrare il server, ricaricarlo se si ferma e così via. Quello che paghi è il numero di CPU, la quantità di RAM e a volte la quantità di traffico di rete usata ogni mese.

La scelta di un provider di servizi specifici per Minecraft è un compromesso tra facilità d’uso e gestione. Questa opzione non ti offre molto controllo sul server; inoltre potresti dover chiedere di caricare dei plug-in, e il servizio potrebbe caricare solo i plug-in noti e collaudati, e non la versione che hai sviluppato tu.

Quello che probabilmente ti serve è invece un VPS (un *virtual private server*, un server privato virtuale). Internet è piena di provider VPS, con vari piani tariffari e package, collocati in diversi paesi e con hardware e funzionalità differenti. Per farti un’idea, cerca in Google “provider VPS”.

Un VPS ti appare come un computer fatto e finito. Puoi connetterti e avere a disposizione tutti i file e il processore. Ti permette di condividere un grosso hardware con altre persone, ma ognuna di esse lo vede come se fosse il proprio computer. Questo significa che come amministratore di sistema puoi installare tutto il software che vuoi; puoi aggiungere account utente, ricaricare e impostare il tuo nome di dominio personale (come `example.com` o `andy.pragprog.com`).

La buona notizia è che puoi lavorare sul server remoto usando la shell della riga di comando, proprio come abbiamo fatto finora, visto che probabilmente il tuo server esegue Linux.

Sistemi operativi remoti

Tutti i servizi VPS offrono virtualmente un unico sistema operativo per il server: Linux. Alcuni provider, come Microsoft con il suo servizio cloud Azure (`http://azure.microsoft.com/it-it/`), può offrire Windows, e qualcuno offre OS X. Sono però soluzioni poco diffuse e relativamente costose. La maggioranza, compreso il servizio cloud di Amazon, lavora con una qualche versione di Linux, un sistema operativo compatibile come POSIX di tipo Unix (POSIX sta per *Portable Operating System Interface*, uno standard per la compatibilità dei sistemi operativi).

FUNZIONALITÀ VPS

Tariffe e funzionalità VPS possono variare di parecchio, ma ci sono alcuni fattori comuni da considerare.

- La maggior parte dei sistemi VPS include una specie di pannello basato sul Web che ti permette di ricaricare il sistema, aggiungere utenti e altre cose del genere. Tra i più famosi c'è `cpanel`.
- Per quanto avere un computer veloce vada sempre bene, per Minecraft è più importante avere a disposizione molta RAM. Procurati un package con quanta più RAM possibile, almeno 2 GB (gigabyte), se non addirittura 4 GB e oltre.
- Mentre scrivevo questo libro, i provider più validi fornivano sistemi a prezzi che andavano da 5 a 50 euro al mese. Se ti sembrano troppi, puoi chiedere ai tuoi giocatori di sostenere il server con una piccola donazione. Attenzione: nel tuo Paese potrebbero esserci delle leggi legate alle raccolte di denaro, quindi chiedi informazioni a qualcuno che conosci nella tua zona prima di avviare un'iniziativa di questo tipo.

Alcuni provider consentono un traffico di rete (*larghezza di banda*) illimitato, mentre altri mettono dei vincoli. Come accade con alcuni piani tariffari del telefono cellulare, se superi il limite di banda, il provider ti farà pagare dei soldi in più.
Altri provider fanno lo stesso con la RAM o lo spazio di memoria: te ne viene concessa una certa quantità, ma se la superi, paghi di più.
Quando confronti i vari piani, assicurati di calcolare anche le eventuali cifre supplementari che potresti dover pagare.

Linux, però, non è solo una cosa. È un insieme costituito dal sistema operativo più altro: il meccanismo a finestre, le utility, i linguaggi di programmazione e così via. Un pacchetto di questo tipo è chiamato *distribuzione*, e ciascuna distribuzione è diversa dall'altra per quello che include, che non include, per come si installa il software e per quando e come si effettuano gli aggiornamenti.

Vediamo alcune delle distribuzioni più diffuse, elencate più o meno in ordine di

popolarità:

- CentOS
- Ubuntu
- Fedora
- Debian
- Linux Mint

Prima di entrare nel panico, tieni presente la cosa più importante e confortante: la shell a riga di comando, `bash`, funziona nella stessa maniera in tutte queste distribuzioni. Tutti i comandi che abbiamo usato in questo libro, come `ls`, `cp` e `cd`, lavorano esattamente nello stesso modo a prescindere dalla versione di Linux che scegli.

Attualmente, Ubuntu e CentOS sono le distribuzioni server più diffuse. La maggior parte dei servizi VPS ti farà scegliere una distribuzione da un elenco ridotto di possibilità.

La differenza principale tra una distribuzione e l'altra sta nella gestione del package: quale strumento si usa per installare i package software e quali package sono inclusi come standard.

Accesso remoto

Per accedere alla shell a riga di comando del tuo server remoto devi utilizzare un insieme di programmi chiamati SSH (da *secure shell*). `ssh` è il comando da eseguire per connettersi al server, il client. Il server esegue `sshd` (da *ssh daemon*) e a questo ti connetti; potrebbe già eseguire un `sshd`, come vedremo tra poco.

Puoi usare `ssh` per connetterti a un server e al suo programma compagno `scp` per copiare i file sul server (che è molto simile al semplice `cp`).

`ssh` deve solo conoscere il nome utente e il nome del server o l'indirizzo IP a cui vuoi connetterti; puoi specificarlo come se fosse un indirizzo e-mail, usando il carattere `@`:

```
$ ssh andy@example.com
Password:
```

Se non hai ancora un nome, puoi usare l'indirizzo IP:

```
$ ssh andy@93.184.216.119
Password:
```

Probabilmente ti è già stato fornito un nome di account, ma se così non fosse puoi crearne uno attraverso l'interfaccia web del tuo provider VPS.

Usa lo stesso tipo di notazione con `scp` per copiare un file. Nell'esempio copieremo un file di testo locale chiamato `myfile.txt` sul server `example.com`, ci collegheremo come `andy` e copieremo il file nella mia cartella *home* (`~/`):

```
$ scp myfile.txt andy@example.com:~/myfile.txt
Password:
```

(Anche in questo caso potresti usare semplicemente l'IP, come in `andy@93.184.216.119`.) Nota che quando esegui `ssh` o `scp` senza fare altro, ti verrà chiesta la password sul computer remoto. Sempre. La cosa diventa presto una seccatura, ma per fortuna esiste un modo migliore per impostare il login remoto.

SSH IN WINDOWS

Gli utenti Windows potrebbero non avere accesso ai programmi a riga di comando ssh e ssh-keygen. In alternativa, possono usare un'applicazione Windows chiamata PuTTY (http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html) per generare le chiavi e connettersi ai sistemi remoti.

Impostare le chiavi SSH

La procedura richiede un paio di passaggi piuttosto semplici. Vedremo i dettagli più avanti. Per ora il procedimento generale è questo.

1. Genera un insieme speciale di chiavi segrete, compresa una "chiave pubblica".
2. Copia la chiave pubblica sulla macchina remota.
3. Accertati che, sulla macchina remota, la chiave pubblica sia protetta.

Fatto questo, puoi eseguire `ssh` e `scp` senza dover fornire una password, e quindi dagli script della shell: è molto comodo quando devi fare una serie di cose sul computer remoto.

Vediamo questi passaggi più approfonditamente.

Per iniziare, accedi alla tua cartella *home* e usa `ssh-keygen` per generare un set di chiavi nello stile RSA (RSA è un sistema crittografico a chiave pubblica):

```
$ cd
$ ssh-keygen -t rsa
```

Quando ti viene chiesta la password, premi Invio; non inserire nessun testo qui. Il comando crea una sottocartella chiamata *.ssh* sotto la tua cartella *home*. Poiché c'è il punto iniziale (.), solitamente non vedi questa cartella elencata come `ls`, ma `ls -a` la mostra:

```
$ ls -a
. .. .profile .ssh (and a bunch of other stuff)
$ cd .ssh
~/.ssh$ ls
id_rsa        id_rsa.pub      known_hosts
```

La tua chiave pubblica è `id_rsa.pub`, e devi aggiungerla al file `~/.ssh/authorized_keys` sul server.

Per farlo, crea prima la cartella *.ssh* sul server. Ti mostro il prompt di login sul server come `Server $`, in modo che tu possa capire su quale macchina mi trovo:

```
$ ssh yourname@example.com
Password:
Welcome to My Awesome Minecraft Server
Last login: Mon Feb 16 12:16:57 from xyzzy-plugh
Server$ mkdir .ssh
```

Sul tuo computer, copia il file `id_rsa.pub` (o come si chiama il tuo file `.pub`) sul server usando `scp`, inserendolo nella cartella *.ssh* e rinominandolo `authorized_keys`:

```
$ scp id_rsa.pub yourname@example.com:~/.ssh/authorized_keys
```

Il file verrà copiato nella cartella *.ssh* sotto la *home* (`~`) e si chiamerà

`authorized_keys`. In seguito, se vuoi, potrai aggiungere in questo file chiavi da altre macchine. Per ora ci basta quest'unica chiave.

Per finire, devi tornare sul server e controllare e sistemare i permessi sui file. La cartella *.ssh* dovrebbe essere leggibile ed elencabile solo da te, e così i file. Nella maggior parte dei casi, `ssh` *non funzionerà* se i permessi non hanno un qualche tipo di limite. È per il tuo bene.

Puoi impostare i permessi sui file con il comando `chmod`:

```
$ ssh yourname@example.com
Password:
Welcome to My Awesome Minecraft Server
Last login: Mon Feb 16 12:26:13 from xyzzy-plugh
Server$ cd .ssh
Server ~/.ssh$ chmod 700 .
Server ~/.ssh$ chmod 600 authorized_keys
```

Ora dovresti poter eseguire `ssh` o `scp` dal tuo computer senza dover specificare una password:

```
$ ssh yourname@example.com
Welcome to My Awesome Minecraft Server
Last login: Mon Feb 16 12:32:07 from xyzzy-plugh
Server$
```

Siamo entrati!

La tua prossima domanda potrebbe essere: "Bene, ma come faccio a uscire?". Puoi disconnetterti premendo Ctrl+D o digitando `exit`; tornerai così alla tua shell locale.

MODIFICARE IL PROMPT

La variabile d'ambiente `PS1` imposta la stringa del prompt per il bash. In questa stringa, puoi usare `\w` per visualizzare la cartella corrente. Quindi, se imposti

```
$ PS1='Server \w\$ '
```

il prompt ti ritornerà come

```
Server ~$
```

Inserisci questa riga nel tuo file di avvio del bash per rendere la modifica permanente (come abbiamo visto nel Capitolo 2).

Amministratori e root

Su un sistema moderno, un utente normale in genere non ha le autorizzazioni per fare qualsiasi cosa. Per avere l'autorità di un amministratore, solitamente devi digitare una password. Sulle macchine Windows questo account è chiamato *Amministratore*, mentre in Linux e OS X si parla di *root*.

Per quanto tu possa connetterti direttamente come root con la password opportuna, la cosa viene vista male. C'è il rischio continuo di commettere qualche errore di digitazione e spazzare via in un attimo metà del sistema. Root è effettivamente potente, ma si sa che da un grande potere derivano grandi responsabilità.

È vero tuttavia che devi poter eseguire alcuni comandi come root senza connetterti effettivamente come root. Per farlo puoi usare `sudo`, che ti permette di agire come farebbe il super utente root. Fai precedere al comando da eseguire la parola `sudo`, e ti verrà chiesta la solita password del tuo account utente normale. Se procedi correttamente, il resto del comando verrà eseguito come root. Per esempio, qui uso `sudo` per eseguire un comando `adduser` che aggiunge l'utente `fred`:

```
Remote $ sudo adduser fred
Password:
User 'fred' added.
```

Puoi così avere il potere pieno di root ma solo in poche circostanze note, senza connetterti come root.

Se il tuo SSH è impostato in modo da consentire la connessione diretta come root, devi cercare di disabilitare questa funzione. In questo modo devi connetterti usando il tuo account utente e una chiave SHH e poi fornire la tua password attraverso `sudo`.

Questo è il metodo più sicuro per permettere l'accesso root solo a te e non ai milioni di malintenzionati che puntano al tuo sistema in ogni momento.

Proteggere l'accesso come root

Può essere che l'impostazione per usare `sudo` sia già attiva. Connettiti al server e prova a eseguire qualche comando semplice (come `id`, che riferisce chi sei) usando `sudo`:

```
Remote $ sudo id
[sudo] password for andy:
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),
4(adm),6(disk),10(wheel)
```

Funziona! Questa macchina è già impostata per `sudo`. Se così non fosse, otterresti un avvertimento:

```
andy is not in the sudoers file. This incident will be reported.
```

Aiuto! È come essere braccati dalla polizia. Ma non temere: il sistema sta semplicemente registrando il tentativo di accedere a un file di registro. Tutto quello che devi fare è aggiungere il tuo nome al file dei permessi per `sudo`.

Connettiti come root e aggiungi la riga che segue in fondo al file `/etc/sudoers` (`nano` è un editor comodo da usare su SSH; vedremo come installarlo e utilizzarlo nel prossimo paragrafo):

```
andy    ALL=(ALL)       ALL
```

Ovviamente dovrai usare il tuo nome, e non `andy`, che è il mio (a meno che non ti chiami Andy anche tu). Salva il file e prova nuovamente a usare `sudo`.

Eseguito `ssh` con successo senza password, e potendo utilizzare `sudo` come root, puoi disattivare l'uso delle password e impedire che root esegua direttamente `ssh`.

La cosa può essere pericolosa, perché potresti inavvertitamente tagliarti fuori dal computer. Per impedirlo, apri una finestra e collegati al server come root. Lascia aperta la finestra e aprine un'altra per iniziare a modificare le impostazioni. Se hai creato un po' di disordine e per qualche ragione non riesci a collegarti o a eseguire `sudo`, avrai sempre questa finestra aperta come root a farti da backup.

Nella nuova finestra, connettiti come root e modifica il file `/etc/ssh/sshd_config`. Dovrai trovare queste due righe e decommentarle, oppure modificarle in modo che riportino `no`:

```
PasswordAuthentication no
PermitRootLogin no
```

Ora devi riavviare il daemon SSH. Su molti sistemi puoi farlo così:

```
Server $ sudo service sshd restart
```

Potresti anche voler modificare la porta predefinita da 22 a un altro numero. Poiché tutti sanno che la porta 22 è per SSH, i malintenzionati bombarderanno il tuo server attaccando da qui. Disattivando il login di root e le password dovresti essere al sicuro, così da avere una chiave pubblica per connetterti, ma se vuoi spostarti altrove per evitare gli attacchi devi procedere diversamente.

Nello stesso file di configurazione (`/etc/ssh/sshd_config`), aggiungi o modifica questa riga:

```
Port 2345
```

scegliendo però un numero diverso da 2345. Ti serve un numero maggiore di

1024, e preferibilmente che non appaia nel file `/etc/services`. Questi sono i numeri che potrebbero essere già utilizzati da altri servizi (compreso 80 per il traffico web).

Quando usi `ssh` per connetterti al server, devi specificare il nuovo numero di porta; puoi farlo attraverso l'opzione `-p`:

```
$ ssh -p 2345 andy@myexample.com
```

Installare i package

La tua distribuzione Linux potrebbe già contenere tutto quello che ti serve, o magari potrebbe mancare qualcosa. Per gestire il materiale, Linux suddivide tutte le utility e i programmi in diversi *package* software. Per esempio, se non lavori con il testo, TeX o Ghostscript potrebbero non interessarti; se non intendi programmare in C o C++, GCC non ti servirà; se non sei su un computer desktop, probabilmente non ti occorre un gestore delle finestre come Gnome o KDE.

Tutti questi package software possono essere installati, eliminati, aggiornati ed elencati usando un gestore apposito (*manager*). Tutte le distribuzioni Linux principali utilizzano un gestore diverso; si potrebbe addirittura dire che è questo gestore a distinguere una distribuzione dall'altra.

I comandi di gestione dei package più comuni sono `yum`, `rpm`, `apt-get` e `deb`. Ognuno ha le sue specificità, ma di base fanno tutti la stessa cosa: scaricare e installare un package software per te.

In particolare, se non hai ancora installato SSH, dovrai installare il package `open-ssh` per poterti connettere al server in modo sicuro. Su Ubuntu, dovrai eseguire un comando del genere:

```
$ sudo apt-get install openssh-server
```

`sudo` esegue il comando come root; `apt-get` è il comando di gestione dei package sotto Ubuntu, e ti viene data la possibilità di installare il package chiamato `openssh-server`.

Su CentOS, il comando è simile, ma usi `yum` invece di `apt-get`:

```
$ sudo yum install openssh-server
```

Analogamente, ti servirà un editor di testi sul server. In realtà non puoi eseguire sul server remoto in Internet un editor completamente visuale (o meglio, puoi se usi X Window System, ma è complicato da impostare e l'esperienza non è così gratificante), quindi avrai bisogno di un editor più semplice come `nano`:

```
$ sudo apt-get install nano
```

Installare Java

A prescindere dal numero di package facoltativi che puoi o meno installare, come minimo dovrai avere Java in esecuzione sul tuo server. Puoi installarlo come abbiamo fatto in precedenza nel libro, ma non dimenticare che ti servirà la versione ufficiale di Java rilasciata da Oracle.

La versione OpenJDK di Java, che potrebbe essere disponibile o già installata sul tuo sistema, è nota per creare problemi con Minecraft; potresti doverla disinstallare usando il gestore di package del sistema. Per esempio, per rimuovere la versione 1.6 da un sistema che usa il Red Hat Package Manager (RPM) esegui questa riga:

```
$ rpm -e java-1.6.0-openjdk
```

Torniamo al buon vecchio Java. Dovrai scaricare il programma di installazione sul tuo computer, quindi copiarlo sul server con `scp`.

Dal tuo computer, vai alla pagina http://www.java.com/en/download/manual.jsp e scorri verso il basso fino alla sezione riguardante Linux. Scarica la versione regolare o quella a 64 bit (dipende dal tuo server) e segui le istruzioni per la tua versione di Linux.

Dovrai poi impostare il tuo script `start_minescraft` e i file JAR sul server come abbiamo fatto localmente nel Capitolo 2.

Eseguire da remoto

Quando esegui uno script come `start_minecraft` mentre sei connesso tramite `ssh`, può sorgere un inconveniente. Quando ti disconnetti, lo script viene eliminato. La cosa non aiuta in un ambiente server.

Fortunatamente, puoi usare un comando chiamato `screen` per far continuare l'esecuzione di Java anche se ti sei scollegato. (Potresti farlo più facilmente anche con il comando `nohup`, ma così perderesti la capacità di immettere direttamente i comandi per il server.) Il comando prende una stringa di argomenti di `-d -m -S` seguita da un nome della tua sessione sul server. Useremo `mcserver` per dare un nome alla sessione; il nuovo script di avvio nello stile del server appare così:

```
#!/bin/bash
cd "$( dirname "$0" )"
screen -d -m -S mcserver java -Xms1024M -Xmx1024M -jar CanaryMod.jar --noControl
```

(Ricorda, `start_minecraft` deve essere eseguibile; accertati di compiere un `chmod +x start_minecraft` se ottieni un errore di "permesso negato".)

Il comando avvierà il server nel suo piccolo mondo, che non verrà influenzato dal fatto che tu ti sia scollegato. Se vuoi "attaccarti" al server in modo da poter eseguire comandi o fare altro, usa `screen` con l'opzione `-r` e il nome della sessione:

```
$ screen -r mcserver
```

Verrai collegato al server con il solito prompt:

```
16:47:49 [INFO] Done (1.163s)! For help, type "help" or "?"
>
```

Per staccarti dalla sessione del server, premi Ctrl+A e poi D. Verrai riportato alla shell originaria, e il tuo server continuerà a lavorare sullo sfondo.

Per vedere i processi in corso, usa il comando `ps`:

```
Server ~$ ps
  PID TTY           TIME CMD
  337 ttys000    0:00.09 -bash
  842 ttys020    0:13.78 /usr/bin/java -jar CanaryMod.jar
```

Per vedere tutti i processi su una macchina, invece, prova `ps ax`. (`ps -?` elencherà le opzioni). In caso di necessità, se dovessi eliminare tutti i processi Java, puoi utilizzare

```
Server ~$ killall java
```

Questa riga chiede a Java di fare pulizia e di uscire in un modo sicuro e affidabile. Se Java si impunta e non obbedisce, passa al più brutale

```
Server ~$ killall -9 java
```

che rimuoverà immediatamente tutti i processi Java, adesso. E stop.

Nome di dominio

Quando imposti il VPS per la prima volta, il tuo provider ti comunicherà l'indirizzo IP del server, cioè il numero che chiunque nel mondo può usare per connettersi. Un indirizzo IP è costituito da quattro serie di numeri separati da punti, come 93.184.216.119 (`example.com`) o 127.0.0.1 (la tua macchina locale, `localhost`).

Volendo, puoi impostare un nome di dominio in modo che gli altri ti trovino per nome (come `pragprog.com` o `example.com`) invece che per numero (93.184.216.119). Il provider potrebbe fornirti questo servizio; se così non fosse, puoi rivolgerti a uno dei numerosi provider in Internet.

I costi annuali per la *registrazione del nome di dominio* vanno in genere da 1 a 20 euro all'anno. La registrazione ti dà il diritto di usare il tuo nome. Avrai anche bisogno di qualcuno che esegua un server di *hosting DNS* che dica che il tuo nome corrisponde al tuo indirizzo IP. Molti registri offrono questo servizio in aggiunta alla registrazione del nome, ma puoi ricorrere a un produttore a parte.

Per scoprire se il nome di dominio che vuoi utilizzare è disponibile, puoi usare lo strumento da riga di comando `whois`. Fai attenzione quando scegli altri strumenti: ci sono in giro tipi sgradevoli che tengono d'occhio le ricerche sui nomi di dominio sul Web e si appropriamo del nome prima di te, provando poi a rivendertelo in cambio di un sacco di soldi.

Come per qualsiasi servizio su Internet, guardati in giro e leggi commenti e recensioni. Alcuni dei provider DNS più famosi e pubblicizzati in realtà hanno una pessima reputazione.

Per continuare

Queste pagine scalfiscono appena la superficie di argomenti che possono diventare molto complessi.

Ci sono un sacco di cose che puoi fare come amministratore di sistema Linux che non abbiamo trattato, come l'esecuzione programmata delle attività, il backup dei dati, l'applicazione di patch di sicurezza e di aggiornamenti, la prevenzione degli attacchi, il perfezionamento delle prestazioni e la configurazione di un server di posta elettronica. Tutte cose divertenti.

Su questi argomenti sono stati scritti libri interi, ma non questo. Spero tuttavia che queste pagine siano sufficienti per farti iniziare.

Promemoria

La lettura di un capitolo ti ha entusiasmato, e decidi di metterti a scrivere del codice, ma all'improvviso ti blocchi: come si fa questa cosa in Java? Non te lo ricordi più. Succede a tutti quelli che imparano un nuovo linguaggio. Una delle prime cose che faccio io è procurarmi o scrivere una specie di promemoria che mi aiuti a ricordare i dettagli.

Linguaggio Java

Quello che trovi qui non è un elenco completo di tutte le caratteristiche di Java; sono solo le parti più importanti che potrebbero servirti e che abbiamo usato nella maggior parte degli esempi del libro.

Tipi di dati letterali

Numeri interi: `int i = 7; i = -5;`

Numeri con decimali: `double num = 3.14; num = 0.01; num = -3e15` (significa -3 volte 10 alla potenza di 15).

Vero o falso: `boolean shouldGetUp=true; shouldGetUp=false` (o qualsiasi espressione booleana che dà `true` o `false`).

Stringa di caratteri di testo: `String s="Hello";`

Array: `String[] grades = {"A", "B", "C", "D", "F", "Inc"};`

Operatori matematici

Somma: `a + b`

Sottrazione: `a - b`

Moltiplicazione: `a * b`

Divisione: `a / b`

Resto: `a % b` (solo per gli integer)

Ordine dei termini: `((a + b) * z)` (come prima cosa vengono sommati `a + b`, poi il risultato viene moltiplicato per `z`).

Operatori di confronto

Uguale a: `a == b`

Diverso da: `a != b`

Minore di: `a < b`

Maggiore di: `a > b`

Minore di o uguale a: `a <= b`

Maggiore di o uguale a: `a >= b`

And: `a && b` (dev’essere qualcosa che è vero o falso).

Or: `a || b` (dev’essere qualcosa che è vero o falso).

Not: `!a` (dev’essere qualcosa che è vero o falso).

Dichiarazioni, importazioni, condizioni, cicli, chiamate

Dichiarare il package: `package name.name.name;`, usando in genere la versione invertita del nome di dominio. Quindi, per esempio un progetto `Wonderful` scritto dai tipi di pragprog.com apparirebbe nel package `com.pragprog.wonderful`.

Importare una classe: `import net.canarymod.api.entity.living.humanoid.Player` dice al compilatore di Java che vuoi usare questa classe. Puoi utilizzare il carattere jolly `*` (come in `inet.canarymod.api.entity.living.humanoid.*;)`, ma così otterresti molte più classi di quelle che ti servono effettivamente.

Dichiarare una classe: `public class` *nome* `{ }`.

Dichiarare una funzione: *nome pubblico del tipo di ritorno (tipi di argomenti e nomi)*, come in `public int myFunction()`.

Dichiarare un blocco di codice: Usa le parentesi graffe `{` e `}`.

Dichiarare variabili: *nome del tipo*; come in `int i;` oppure, a uno `Stack<Location> myLocations;` generico devono essere aggiunti i tipi specifici racchiusi tra le parentesi angolari `<` e `>`.

Assegnare valori alle variabili: `a = 10; s = “Bob”; x=3.1415;`.

Prendere decisioni: Decidi quale codice eseguire usando un `if` (la parte `else {}` è facoltativa):

```
if (true or false comparison) { // è vero  doThis();} else { // è falso  doOtherThing();}
```

Cicli: Ci sono tre modi per eseguire un ciclo in un blocco di codice.

- Usare il costrutto `for-each`: `for(`*variabile di tipo* `:` *collezione*`)`. Esempio: `for (Player player :playerList) {` *blocco* `}`
- Usare il ciclo `for`: `for (int i=0; i<` *limite*`; i++) {` *blocco* `}`
- Usare il ciclo `while`: `while (`*qualcosaÈVero*`) {` *blocco* `}`

Creare un nuovo oggetto: `Foo thing = new Foo();`, dove `Foo` è il nome della classe e `thing` è la variabile da assegnare al nuovo oggetto.

Chiamare una funzione: Usa le parentesi, come in `getServer()`, che restituisce un oggetto `Server` per un plug-in.

Modificatori della visibilità

final: Non permettere a nessuno (te compreso) di modificare questo valore una volta che è stato impostato. È quello finale, definitivo.

static: Mantieni questi dati o questa funzione disponibile al di fuori di un dato oggetto. Se necessario, puoi mescolare `final` e `static`.

public: Chiunque può vedere e utilizzare questa funzione o questi dati.

protected: Questa classe e le altre classi in questo package possono vedere e usare questa funzione o questi dati.

private: Solo questa classe può vedere e usare questa funzione o questi dati; le sottoclassi e le altre classi nello stesso package non possono farlo.

Poiché i modificatori `public` o `private` si usano per ciascuna dichiarazione, il codice che segue crea una costante visibile pubblicamente e non modificabile all’esterno di qualsiasi oggetto:

```
public static final int pi = 3.1415;
```

Conversioni di tipi di dati

Da int a double: Assegna: `int now = 72; double temperature = now;`.

Da double a int: Esegui il cast; la parte con i decimali viene eliminata: `double when = 15.375; int day = (int)when;`.

Da String a int: Usa `parseInt`: `int foo = Integer.parseInt(“365”);`.

Da String a double: Usa `parseDouble`: `double foo = Double.parseDouble(“3.1415”);`.

Da double a String: Usa `valueOf`: `String.valueOf(3.1415);`.

Da int a String: Usa `valueOf`: `String.valueOf(72);`.

Da Double a String: (La classe `Double`, non la primitiva) `Double.toString(3.1415);`.

Da Integer a String: (La classe `Integer`, non la primitiva) `Integer.toString(72);`.

La concatenazione delle stringhe verrà convertita automaticamente, quindi

`String s =` *“La temperatura è di”* `+ 72 +` *“gradi.”*

darà la stringa

“La temperatura è di 72 gradi.”

Istruzioni di importazione comuni

Nelle prossime pagine trovi un elenco di tutti i package e dei nomi di classe importati che abbiamo usato nei plug-in del libro.

Se ottieni un errore del tipo “simbolo non trovato”, potresti dover importare il nome della classe completa. Per esempio, per un’`HashMap` sarà `import java.util.HashMap`, per un `Block` sarà `import net.canary-mod.api.world.blocks.Block`.

Puoi sempre recuperare i nomi completi delle classi nella documentazione di Java o di Canary, ma per tua comodità ti elenco qui le più comuni.

```
import com.pragprog.ahmine.ez.EZPlugin;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.Iterator;
import java.util.List;
import java.util.Map;
import java.util.Stack;
import net.canarymod.BlockIterator;
import net.canarymod.Canary;
import net.canarymod.LineTracer;
import net.canarymod.api.DamageType;
import net.canarymod.api.entity.Entity;
import net.canarymod.api.entity.EntityType;
import net.canarymod.api.entity.living.EntityLiving;
import net.canarymod.api.entity.living.animal.Bat;
import net.canarymod.api.entity.living.animal.Cow;
import net.canarymod.api.entity.living.animal.Squid;
import net.canarymod.api.entity.living.humanoid.Player;
import net.canarymod.api.entity.living.monster.Creeper;
import net.canarymod.api.factory.EntityFactory;
import net.canarymod.api.factory.PotionFactory;
import net.canarymod.api.inventory.ItemType;
import net.canarymod.api.potion.PotionEffect;
import net.canarymod.api.potion.PotionEffectType;
import net.canarymod.api.world.Chunk;
import net.canarymod.api.world.World;
import net.canarymod.api.world.blocks.Block;
import net.canarymod.api.world.blocks.BlockType;
import net.canarymod.api.world.blocks.Sign;
import net.canarymod.api.world.effects.Particle.Type;
import net.canarymod.api.world.effects.Particle;
import net.canarymod.api.world.effects.SoundEffect;
import net.canarymod.api.world.position.Location;
import net.canarymod.api.world.position.Vector3D;
import net.canarymod.chat.MessageReceiver;
import net.canarymod.commandsys.*;
import net.canarymod.database.Column.DataType;
import net.canarymod.database.Column;
import net.canarymod.database.DataAccess;
import net.canarymod.database.Database;
```

```
import net.canarymod.database.exceptions.*;
import net.canarymod.hook.HookHandler;
import net.canarymod.hook.entity.DamageHook;
import net.canarymod.hook.entity.ProjectileHitHook;
import net.canarymod.hook.player.ItemUseHook;
import net.canarymod.hook.player.TeleportHook;
import net.canarymod.hook.world.ChunkCreatedHook;
import net.canarymod.hook.world.ChunkLoadedHook;
import net.canarymod.hook.world.ChunkUnloadHook;
import net.canarymod.logger.Logman;
import net.canarymod.plugin.Plugin;
import net.canarymod.plugin.PluginListener;
import net.canarymod.tasks.ServerTask;
import net.visualillusionsent.utils.PropertiesFile;
```

A virgola mobile. Un numero con una parte decimale, come 1,25.

Annotazione. Un comando aggiunto al codice sorgente di Java che modifica o aggiunge informazioni (come una funzione di tag o un gestore di eventi).

Argomento. Un valore passato a una funzione, che lo utilizzerà.

Array. Un elenco sequenziale di valori a cui si può accedere in base a un indice.

Array. Classe Java che implementa un array.

Attività. Un segmento di codice eseguito in un thread che ha uno scopo ben definito.

Blocco. Un elenco di istruzioni di codice delimitato dalle parentesi `{` e `}`.

Booleano. Un valore logico che può essere solo `true` (vero) o `false` (falso).

Cartella corrente. La cartella (in inglese *directory*) di lavoro attuale.

Cast. Serve per modificare l'interpretazione di un valore, solitamente da un oggetto a un tipo particolare di genitore.

Classe. Una "ricetta" che dice al compilatore come creare un oggetto, cioè quali dati e funzioni dovrà contenere.

Client. Un software eseguito da un utente e dotato in genere di un'interfaccia grafica. Si connette a un server.

Codice sorgente. Le istruzioni in linguaggio Java che si digitano in un file.

Compilazione. Prende un file di testo di istruzioni leggibili dall'uomo e lo converte in qualcosa di eseguibile da un computer (solitamente in un formato binario). Il compilatore di Java è javac.

Costruttore. Una funzione nella definizione di una classe che viene chiamata quando si crea un nuovo oggetto. Puoi usarlo per impostare le variabili degli oggetti e altro.

Distribuire. Installare una risorsa in un ambiente server.

DNS (*Domain Name Service*). È il sistema globale che traduce un nome di dominio come `esempio.com` in un indirizzo IP come 93.184.216.119.

Double. Un grosso numero a virgola mobile.

Eccezione. Un errore che interrompe la funzione in corso e avvia la funzione di `catch` (cattura) più generale o che blocca un programma se è operativo.

Ereditare. Utilizzare qualcosa proveniente da un genitore.

Eseguibile. Un file eseguibile dal computer, in genere di tipo binario, con istruzioni macchina di basso livello; a volte è uno script di testo eseguito da una shell.

Evento. Un oggetto che rappresenta un'azione del mondo reale, come un clic del mouse.

File binario. Un file che contiene numeri binari e che non è leggibile dall'uomo.

File di testo. Un file che contiene caratteri di testo leggibili dall'uomo.

File system. L'insieme organizzato dei file e delle cartelle presenti sul computer.

final. Parola chiave di Java che indica che quella variabile non può essere modificata.

Float. Un numero a virgola mobile non grande come un double.

Funzione. Un elenco di istruzioni Java dichiarato con un tipo di ritorno, un nome, i parametri che prende e un blocco di codice racchiuso tra `{` e `}`.

Hash. Una lista di oggetti indicizzata in base a un tipo qualsiasi di oggetto (in genere una stringa).

HashMap. Classe Java che implementa un hash.

I/O (*Input/output*). Inviare e ricevere dati da un punto che si trova altrove, come un file o una rete.

import. Una parola chiave di Java che permette di usare una classe di un altro package, come `java.util.HashMap` (che è la classe `HashMap` nel package `java.util`).

Integer. Un numero intero senza decimali, come 7.

Iteratore. Un oggetto che consente di recuperare un valore alla volta da un determinato tipo di collezione (come un `ArrayList` o un'`HashMap`).

JAR. Un file di archivio Java che contiene i file `.class` e i file di configurazione.

Letterale. Un valore che viene digitato direttamente, come `123`, `true` o `"Notch"`.

Listener. Una funzione che viene chiamata quando accade qualcosa di interessante.

localhost. Il nome proprio del computer in rete.

long. Un numero intero molto grande, senza decimali.

Mappare. Vedi *HashMap*.

null. Una variabile che non punta a nulla e viene impostata al valore speciale `null`.

Oggetto. Un insieme di variabili e funzioni “vive” create da una classe-ricetta.

Orientato agli oggetti. Software basato sulla teoria degli oggetti che combina variabili e funzioni in una pila di dati.

Package. Un insieme di classi Java coordinate tra loro.

Parametro. Un valore passato a una funzione.

Parola chiave. Una parola definita da Java come parte del linguaggio. Le parole chiave non possono essere modificate, né si possono usare i loro nomi come variabili.

Percorso. Un elenco di cartelle tra le quali il computer cercherà per trovare un comando o altre risorse.

Plug-in. Un segmento di codice compilato che viene aggiunto a un altro segmento di codice già compilato.

Porta. Un numero convenzionale che permette ai computer di comunicare in rete. Per esempio, il Web usa la porta 80, mentre Minecraft usa la porta 25565.

private. Parola chiave di Java che limita la visibilità alla classe corrente.

public. Parola chiave di Java che estende la visibilità a tutte le classi.

Riga di comando. La finestra del terminale dove digitare i comandi.

Script. Un file di testo che contiene comandi della shell (o di un altro linguaggio testuale, come Ruby o Python).

Server. Un software eseguito in background, in genere su un altro computer, che può servire più connessioni client. È anche chiamato *macchina remota*.

Shadowing. Quando una variabile ha lo stesso nome di un’altra variabile nel suo ambito, si dice che “le fa ombra” (*shadow*). Potenzialmente può creare dei problemi.

Shell. Vedi *Riga di comando*.

Simbolo. Un insieme di caratteri leggibili dall’uomo che significano qualcosa di speciale per il compilatore Java come parte del linguaggio di programmazione.

Statica. Una variabile o funzione che non si trova in alcun oggetto specifico.

Stringa. Un insieme di caratteri leggibili dall'uomo contenuti in una variabile.

Thread. Un elenco di funzioni eseguite in ordine dal computer. I thread possono essere interrotti da altri thread.

Tick. Una unità di tempo arbitraria. Un tick del server di Minecraft è pari a 1/20 di secondo.

Variabile. Un contenitore di dati che ha un nome. Può essere un valore immediato come 15 o può puntare a un oggetto come un `Player` o una `Cow`.

Variabile d'ambiente. Le impostazioni usate dalla shell e dai programmi applicativi.

Variabile globale. Una variabile che può essere utilizzata da più funzioni e/o classi.

Variabile locale. Una variabile dichiarata con un blocco di `{` e `}` che può essere utilizzata solo all'interno di quel blocco.

void. Niente. Una funzione dichiarata in modo che restituisca `void` non fornirà alcun valore e non necessita di un'istruzione di ritorno.

VPS (*Virtual Private Server*). Un computer remoto che si può affittare al mese o in base alla quantità di CPU e traffico di rete che si usa.